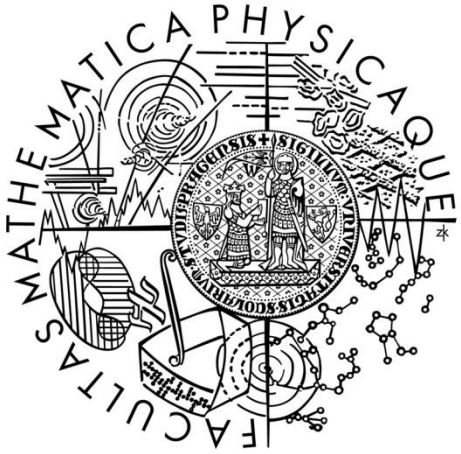
Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS



Filip Pavliš

Analysis of Knowledge Obsolescence in Ensemble-Based Component Systems

Department of Distributed and Dependable Systems

Supervisor of the master thesis: doc. RNDr. Tomáš Bureš, Ph.D.

Study programme: Computer Science

Specialization: Distributed and Dependable Systems

Prague 2015

I would like to express my thanks to my supervisor doc. RNDr. Tomáš Bureš, Ph.D. for guiding me through this Master thesis. I would like to thank my family, especially my father and mother, for their support. Last but not least, I would like to thank Charlota for all her love, support and food!

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, 4th May 2015 signature of the author

Název práce: Analysis of Knowledge Obsolescence in Ensemble-Based Component Systems

Autor: Filip Pavliš, email: f.pavlis@gmail.com

Katedra: Katedra distribuovaných a spolehlivých systémů (KDSS)

Vedoucí bakalářské práce: doc. RNDr. Tomáš Bureš, Ph.D., KDSS

Abstrakt

Návrh distribuovaných vestavěných systémů je často netriviální proces. Jedna z potřeb je zaručení korektního chování systému. To vyžaduje ověřit, že informace propagované skrze systém jsou spolehlivé tedy především aktuální. Cílem práce je výzkum a implementace analýzy, která bude schopna identifikovat zastarávání hodnot, které vzniká z důvodu zpoždění při rozvrhování a komunikaci v systémech reálného času. Analýza bude navržena pro Ensemble-Based Component System (EBCS) sémantiku, která umožňuje formální specifikaci vstupu a zároveň je dostatečně obecná. Hlavním úkolem bude nalézt správný vstupní model analýzy a její možné limity. Úsilí by mělo být vloženo do balancování mezi úrovní abstrakce, kterou poskytneme uživateli, a sílou analýzy jako takové. Hlavním přínosem analýzy bude detekce situací, při nichž jsou data, která jsou zpracovávána v systému, zastaralá a mohou způsobit nevhodné chování jednotlivých komponent.

Klíčová slova: časová analýza proměnných, analýza zastarávání hodnot, EBCS, systémy reálného času, zpoždění ve vestavěných systémech

Title: Analysis of Knowledge Obsolescence in Ensemble-Based Component Systems

Author: Filip Pavliš, email: f.pavlis@gmail.com

Department: Department of Distributed and Dependable Systems (D3S)

Supervisor of the master thesis: doc. RNDr. Tomáš Bureš, Ph.D., D3S

Abstract

Designing Resilient Distributed embedded Systems is a challenging task. One of the design issues is to guarantee correct behavior of the system during the runtime. It demands verification that information propagated through the system is reliable. The goal of this thesis is a research and implementation of an analysis that should identify obsolescence of variables due to delays caused by scheduling and communication in real-time systems. Analysis will be designed for Ensemble-Based Component System (EBCS) semantics because it enables precise specification and analysis of important properties. The main problem is to find a suitable input model of the analysis and find its possible limits. Effort should be put in balancing between the level of abstraction given to a RDS developer and power of the analysis itself. The main benefit of the analysis will be detection of situations in which data processed in RDS are outdated and can cause incorrect behavior of particular components.

Keywords: timing analysis, knowledge obsolescence analysis, knowledge field obsolescence, Ensemble-Based distributed systems, DEECo, real-time system, delays in component systems

Contents

Chapter 1

Introduction

Many different components (devices) currently require communication and data interchange by forming large-scale Resilient Distributed Systems (RDS). These systems also react to and affect their environment which provides a need for their greater autonomy.

Ensemble-Based Component Systems (EBCS) [1] [2] form a class of component based systems which are suitable for designing RDS. Components of EBCS are bonded together via ensembles interchanging their values by forming dynamic groups. Because of the dynamism, components introduce autonomic and self-adaptive behavior.

DEECo [1] [3] [4] is a component model which embodies EBCS concepts and gives them suitable semantics which allows its usage in RDS engineering. DEECo has framework implementation for Java, C++ and other languages.

When engineers need to guarantee correctness or even safety of RDS they need to prove that each critical task of such RDS returns correct results for given inputs in a given deadline. This involves proving that all tasks are schedulable. Schedulability analysis is a well-covered research topic in real time systems. However, there is one hidden and very complex problem. Are input values of real time tasks fresh enough? Or is it possible that some sensor affected by a long period of time keeps some outdated value flowing around the system? This question is hard to answer without a further simulation or analysis, especially in complex dynamic systems.

The main target of this thesis is to answer the proposed question by an analysis whose input will be defined via the DEECo semantics. The semantics is formally defined, is simple but also very flexible for the modeling of EBCS and the usage of the semantics as an input of the analysis helps to decrease its complexity. A simulation is not the target of the thesis. There is a simulation project for DEECo that is created in parallel [5].

Let us introduce a term *belief*. The DEECo abstraction formalizes the term *belief* more precisely and we introduce it properly in Chapter 2 with DEECo formalization. By the term *belief* we identify a value of some variable the system works with. Each value is calculated from other variables or measured by sensors; this means it is not always precise because the outside environment keeps changing over time. In fact each system never works with current data but with belief which is always shifted from the real value. However situations can occur when the belief is too outdated which can negatively affect the results of the system. For instance a collision detection system can cause a crash having obsolete information about a distance from an obstacle.

The goal of this thesis is to design an analysis tool that evaluates how the *belief* of a particular variable is outdated from the "reality" respective to real knowledge field of which belief it is. We identify the delay of a knowledge field by the term *obsolescence*. It will be the user's task to define which belief the system works fine and what is considered as incorrect behavior. To be able to do a more precise reasoning about a particular system we will design the analysis over the DEECo abstraction. The main benefit of DEECo is that it allows us to analyze the *obsolescence* of variables over formalized, simple and abstract semantics which is also expressive enough to describe a wide range of RDS.

Chapter 2

Running example and DEECo concepts

In this chapter we are going to introduce a running example that will help us analyze important questions on a practical example. We will also present the DEECo semantics overview and use it to formalize the example.

2.1. Running example

For the past few years there has been a lot of innovations in autonomous vehicles. This presents ideas about smart cities where traffic could be controlled by computers. This concept would help optimize traffic and also lead to lower expenses for public transportation. If all vehicles are equipped with such control units it will be natural to provide them with distributed communication forming large scale RDS. This can for instance bring a smarter version of an autonomous cruise control system where the primary input would not be from sensors but from the distributed communication between the vehicles. In our example we focus on the autonomous cruise control system.

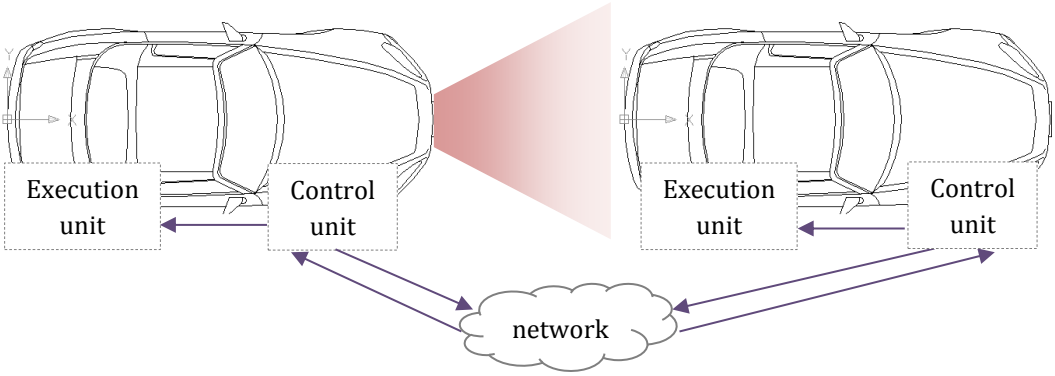


Figure 2.1: Illustration of the main units of the autonomous cruise control system

Let us introduce a very simplified specification of the autonomous cruise control system with a distributed communication involved. The specification is depicted in the Figure 2.1 above.

There is a *control unit* that is responsible for the interchange of values between the vehicles and making decisions about the driving plan. If the followed car is getting close then the *control unit* stops accelerating or even starts to brake. On the other hand if the followed car is getting further the *control unit* starts to accelerate. These commands are then sent to something more abstract called an *execution unit*. It does not matter for this example what the unit does but we need it to include communication delays between the *control unit* and the final execution.

The figure also shows sensors which are necessary for the system to be able to detect pedestrians or unexpected obstacles. We are not going to cover these sensors because we are primarily focused on delays caused by the communication.

To be able to define the analysis input we provide basic information about the system such as what processes it is composed from, what its scheduling aspects are, what the basic data flow is and how it is affected by communication delays.

Because we are doing an analysis not a simulation, we can evaluate only some particular state of the system. We cannot evaluate whole driving experience from one place to another. Because of this we introduce a scenario where one vehicle follows the other one; we are interested in a safe distance between the vehicles. We would like to know if the following car will be able to brake safely when the followed car decrease its speed rapidly.

In the next step we need to define the example via DEECo semantics but before that we introduce the semantics overview.

2.2. DEECo overview

In this chapter we present the overview of the main parts of the DEECo semantics taken from [3] where formal definitions can be found.

Knowledge field is basically a variable. It is organized as a hierarchical data structure mapping knowledge identifiers to values. Specifically, values may be either potentially structured data or executable functions.

Component is a unit which contains a set of knowledge fields and processes. Each process works with knowledge fields from the component it belongs to. Formally $C = (K_C,\ P_C,\ I_C)$, where K_C is a set of knowledge fields. P_C is a set of processes and I_C is the initial knowledge valuation.

Knowledge valuation V_C is a partial function $V_C: K_C \rightarrow D$, where D denotes the domain of knowledge field values.

Process p of component C is technically just a function that computes a new valuation of particular knowledge fields of the component C and preserves the knowledge valuation of the remaining fields.

Belief refers to the part of the component's knowledge that represents a copy of knowledge of another component, and is thus treated with a certain level of uncertainty as it might become obsolete or invalid. Imagine we have a sensor measuring the distance from an object. The measured distance is then transferred from component A to another component B. The knowledge which we have accessible from component B is our belief or copy of the original knowledge located in component A. In DEECo we assume that each component is associated with an arbitrarily outdated belief of knowledge valuation of any other component in the system. This helps us to model the distributed communication independently on used middleware with relatively few restrictions. Belief is propagated between each pair of components in the system via a mechanism called belief propagation.

Ensembles serve as "communication pipes" between components where one component always plays the role of the ensemble's coordinator while the others play a role of a member. This is determined dynamically according to the membership predicate of the ensemble. As to the interaction, the individual components in an ensemble are not capable of explicit communication with others. Each ensemble in an application is an instance of an ensemble definition, formalized as follows.

For components C_i and C_j we define ensemble as a tuple $E = (B_E, M_E)$, where B_E denotes the membership predicate defined over knowledge valuations of those components, and M_E denotes the mapping function, which computes the update of the component's knowledge as a result of knowledge exchange implied by the ensemble.

Ensembles run locally which means that they do not interchange any values over the network. This can be achieved thanks to the concept of belief propagation where each component has belief about other components locally accessible. Components cannot directly access their belief. It is accessible only to ensembles that can update the component's ordinary knowledge fields.

System is a tuple $S = (\mathbb{C}, \mathbb{E})$, where $\mathbb{C}$ represents the set of all components and $\mathbb{E}$ denotes the set of all ensemble definitions.

2.3. DEECo execution semantics

In the next section, we introduce a runtime semantic of DEECo via automata. This is necessary to identify which units can run in parallel and what timing properties the system may have. The overview we present here was taken from [3].

Process is an activity local to a component that atomically reads a subset of a component's knowledge, performs a computation on it, and updates the component's knowledge with the result of the computation. To model this, we associate each process $p \in P_C$ of each component C with an automaton $A(p)$ – depicted in the figure below.

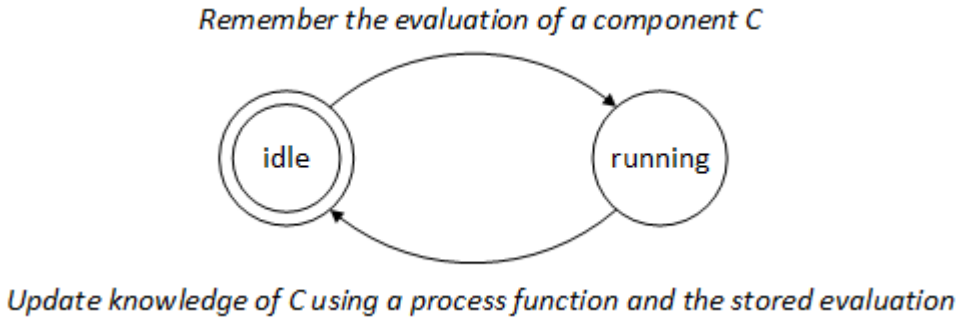


Figure 2.2: Automaton representing process

Belief propagation. To model the belief propagation, we associate a FIFO queue $Q_{C_i}^{C_j}$ with each ordered pair of components C_j, C_i, $C_j \neq C_i$. The queue will have two operations, *enqueue* which inserts the current knowledge valuation of C_j into the queue and *dequeue* which removes the valuation and assigns it as the current belief of C_i regarding knowledge valuation of C_j. Then we associate each such queue with an automaton $A(Q_C^{C_j})$, depicted in Figure 2.3, which repeatedly and non-deterministically performs the *enqueue* and *dequeue* operations.

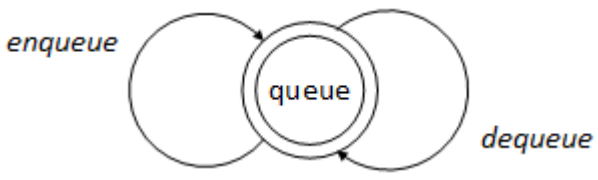


Figure 2.3: Automaton representing belief propagation

Ensemble. For each ensemble $E \in \mathbb{E}$ we define, automaton $A\left(E_{co}^{(C_i,C_j)}\right)$ for the coordinator (depicted in Figure 2.4) and automaton $A\left(E_{mem}^{(C_i,C_j)}\right)$ for the member (depicted in Figure 2.5).

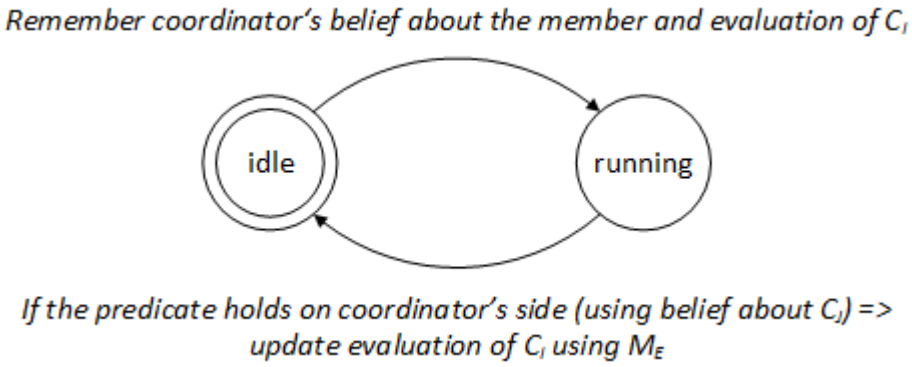


Figure 2.4: Coordinator's side of an ensemble automaton

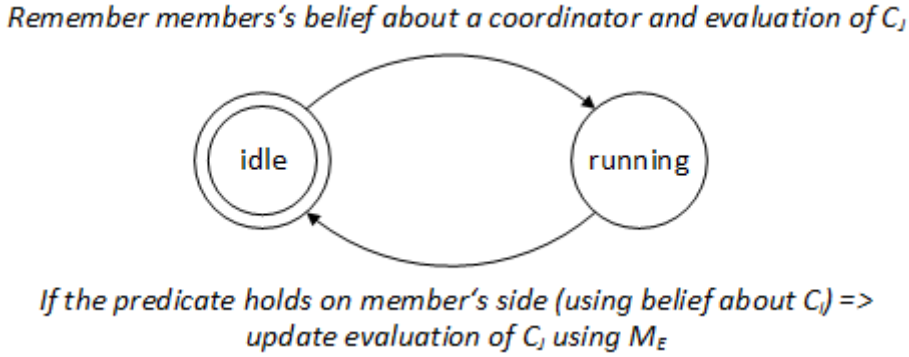


Figure 2.5: Member's side of an ensemble automaton

We define automaton $A\left(E^{(C_i,C_j)}\right) = A\left(E_{co}^{(C_i,C_j)}\right) \times A\left(E_{mem}^{(C_i,C_j)}\right)$ corresponding to a potential ensemble (based on the ensemble definition E) in which C_i takes the role of the coordinator and C_j takes the role of the member. The automaton $A(E)$ for ensemble E is then defined as follows.

$$A(E) = \prod_{\forall\, C_i, C_j \in \mathbb{C}} A(E^{(C_i,C_j)})$$

System. The final automaton for DEECo system $S = (\mathbb{C}, \mathbb{E})$ is defined as follows.

$$A(S) = \prod_{\forall\, \mathrm{C} \in \mathbb{C}} \prod_{\forall\, \mathrm{p} \in P_c} A(p) \times \prod_{\substack{\forall\, C_i, C_j \in \mathbb{C} \\ C_i \neq C_j}} A(Q_{C_i}^{C_j}) \times \prod_{\forall\, E \in \mathbb{E}} A(E)$$

2.4. Timing aspects of the semantics

In this chapter we introduce timing aspects of the DEECo semantics. The formal definition of timing aspects of DEECo can be found in [3]. We associate a timestamp to each action in an execution trace generated by transitions of the automaton $A(S)$.

Periodic process semantics is the same as in traditional embedded systems – the process executes periodically (once in each period) and has a relative deadline equal to the period. To account for different initial arrival times, the first period of each process starts at a predefined offset.

Triggered process p of a component C is associated with a set of knowledge fields $G_p \subseteq K_C$ upon whose change it is triggered. Further, the triggered process is associated with a relative deadline D, which serves as the upper bound of time between the knowledge change and the finish of the corresponding process iteration.

Periodic ensemble is similar to a periodic process. In ensemble, it means that the membership predicate and the mapping function are executed periodically but independently for the coordinator and members of the ensemble.

Triggered ensemble is similar to a triggered processes, it executes whenever any knowledge (or belief) triggers it on change. We thus associate the definition of a triggered ensemble E with a set of knowledge fields $G_{C0} \subseteq K_{C_{C0}}$ (for the coordinator) and $G_{mem} \subseteq K_{C_{mem}}$ (for the member) upon whose change it is triggered. In the same way as a triggered process we associate a triggered ensemble with a relative deadline D.

Belief propagation. The communication queues introduced earlier contain the enqueue and dequeue transitions which model the latency of network stacks of both peers and latency of the network itself.

2.5. Running example via DEECo semantics

Because we have introduced the DEECo semantics in the previous chapter we can now define the running example via the DSL (Source code 2.1). In next chapters we introduce algorithms that are capable of analysis of such input.

```
1.  component Vehicle:
2.      knowledge:
3.          distance (from the followed vehicle)
4.          speed
5.          followedVehicleSpeed
6.          position
7.          engineInstructions
8.      process ControlUnit:
9.              in distance, in speed, in followedVehicleSpeed, out engineInstr…
10.         function: …
11.         scheduling: periodic(400ms)
12.     process ExecutionUnit:
13.             in engineInstructions
14.         function: …
15.         scheduling: periodic(500ms)
16. ensemble UpdatePositionInformation:
17.   coordinator: Vehicle
18.   member: Vehicle
19.   membership: inRange(coordinator.position, member.position)
20.   knowledge exchange:
21.     coordinator.distance.update(member.position);
22.     coordinator.followedVehicleSpeed.update(member.speed);
23.   scheduling: periodic(1000ms);
24. system.beliefPropagationDelay = 200ms;
```

Source code 2.1: The running example defined via the DEECo DSL

Chapter 3

Analysis

In this chapter we are going to analyze various design aspects of the main algorithm of the analysis tool. This will include discussion about required inputs and expected output. We will also introduce simplifications of DEECo. The analysis will be covered by the running example described in the previous chapter.

3.1. Main Requirements

We need to analyze the *obsolescence* of knowledge fields of a system defined by the DEECo semantics. We would like to design analysis with the least possible number of constrains and be able to run the algorithm in polynomial time.

We are not going to dive into a schedulability analysis since there was a lot of work done in this field and it is very complex topic [6]. We may encounter terminology conflicts since schedulability analysis is often referred to as timing analysis or safety analysis, but these terms are more general and schedulability analysis is just their subset.

Let us give a basic overview of the main steps of the algorithm. First we find the maximal delay during which the field is not updated. Then we compute possible worst case estimations of real world values in a scenario when everything is going great (i.e. lower bound) and in a scenario where everything is going bad (i.e. upper bound) for the given delay. The terms great and bad are only for illustration purposes.

We can imagine a whole system as a set of knowledge fields which are bound together via processes and ensembles. Basically every knowledge field is known from the beginning or calculated from another one via process or ensemble. Processes and ensembles are very similar in both respect to data flow and scheduling. The only specific difference occurs in the case of dynamic ensembles where it is onwards unknown which components will be members of the interchange since it is directly affected by runtime values. If we omit dynamic ensembles for a while, we can say that ensembles and processes are functions which take some knowledge fields as input and update other knowledge fields with results. Each function would also have a period and a relative deadline.

To make our evaluation algorithm less complex we support only primitive data types, which means that the user is not able to define structured data types like classes or structures. This will also simplify the analysis and allows us to avoid complications when each individual part of a structured type can be calculated by different processes and affected by different delays.

A component will be perceived only as an organizational unit that serves as a namespace for knowledge fields and processes. It will have no direct impact on the analysis. We only have to keep in mind that a process can access only knowledge fields from the same component. An ensemble is a communication bridge between components.

3.2. Analysis of desired analysis outputs

The first question we should answer is: What type of output do we expect to obtain from the analysis? Would the maximal delay of values be sufficient? Or should we require some lower and upper bound values estimates?

As we described in the previous chapter the system is just a set of knowledge fields bound together by ensembles and processes. If we focus on our running example, we can be in a situation where we calculate the current distance from the current speed. We may be interested in the *obsolescence* of a knowledge field representing the distance.

Let us say the solution we would choose is that the analysis would return delays only. What can we say about the distance field? If the process calculating the field has a delay equal to d_p then we can say that the distance field is affected by the delay of $d_p\ ms$. However the speed knowledge field may also be affected by a delay, for instance the delay of the ensemble that transfers it, let us say it is $d_e\ ms$. The analysis then cannot simply return a delay equal to $d_p + d_e$ because it says nothing about the real value of the distance field. Return delay equal to d_p is also wrong. The reason is the speed knowledge field was affected by the delay d_e which means that it directly affects the *obsolescence* of the distance which is calculated from the speed. Even if we say we are satisfied with sum of those delays, a problem occurs when we have more input values and they are all affected by different delays. Using this way the analysis would be very imprecise and would not have useful output.

If we choose a second solution and evaluate worst values estimates we can simply pass the estimates of the speed knowledge field to the process function, that calculates the distance, and get estimates of the distance knowledge field.

In the previous chapter we have demonstrated that we need to find estimates of obsoleted knowledge field values. This means that we need to calculate something like upper and lower bounds in which the real values will oscillate. We cannot say how often these bounds will be reached in practice but our goal is to find worst cases. We can demonstrate it on the running example. If the followed car accelerates as much as possible during the whole measured period, then the distance between the vehicles is the greatest possible (upper bound). Vice versa if the followed brake as much as possible then the distance is the lowest possible (lower bound). In practice these situations will not be that common. It can happen that we actually never reach those values in reality. However, we do not care about average situations because our analysis serves as a tool for getting some guarantees.

If we summarize the previous chapters we can imagine a whole system as a set of knowledge fields bounded together by oriented edges. These edges represent processes and ensembles which do transformation of upper and lower bounds (estimations) from input fields to output fields. This is the main technical design which arose from our analysis of requirements.

3.3. Identifying delays

In the previous chapter we introduced the evaluation of the *obsolescence* of knowledge fields based on the delay determined by timing aspects of the processes that calculate them. But what exactly is the delay of a process in respect to its output knowledge field?

The delay of a knowledge field *K* that is the output of a process *A* is a maximal amount of time during which the value of the knowledge field *K* is not updated by the process *A* and during which it can be used as an input of another process. To be able to calculate the delay, we distinguish between two types of processes (or ensembles). The first type is represented by processes whose output field is used by another process or ensemble, we call them *producers*. The second type represents processes whose outputs field is not used anymore, we call them *consumers*. In the running example a *producer* is the control unit process and the *consumer* is the execution unit process.

First we calculate a delay of the first type of processes - *producers* whose output is used by other process or ensembles. In Figure 3.1 we can see the schedule of a sensor process. The process starts after some time and then reads the fresh value from sensors does its calculations and returns the output. Our question is how long is the output value not updated by the sensor process?

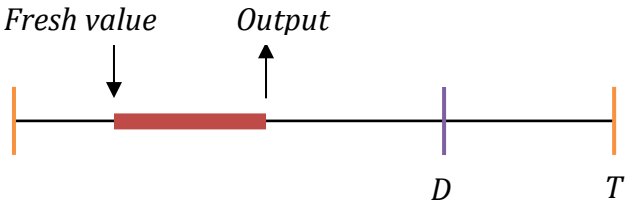


Figure 3.1: The Schedule of a sensor process. T denotes a period. D denotes a relative deadline. A red box denotes the execution time of the process.

On the timeline depicted in Figure 3.1 it may look like the delay is equal to the period T. Because of this we inspect what schedule configuration may occur after the second run of the process. In such configuration we let the process run at the beginning of the first period and close to the end of its relative deadline in the second period. Such scenario is illustrated in Figure 3.2.

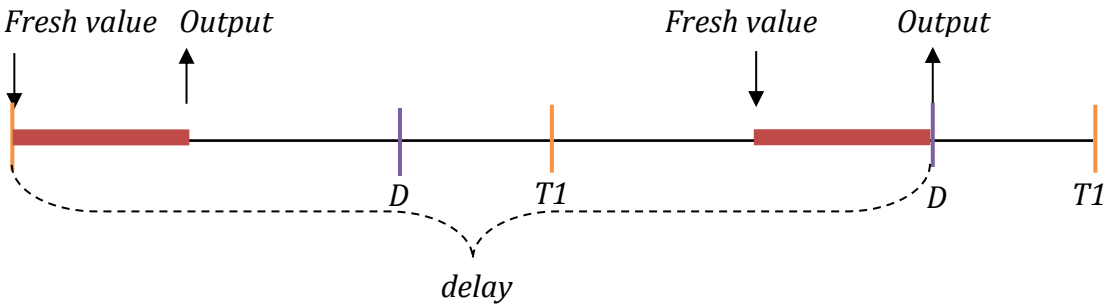


Figure 3.2: A schedule of a process executed in two periods. The process is planned to the beginning of the first period and close to its relative deadline in the second period.

What we find is that the real possible delay can be equal to $T + D$. This means that other process that use this knowledge field may read it in time equal to $T + D - \varepsilon;\ \varepsilon \to 0^{+}$. That means that such a process would work with a value that is approximately $T + D$ old.

Now let us inspect the *consumer* process type. The approach used above for a *consumer* process violates the second part of the delay definition because the output is never used by any other process. What makes sense is to compute how long the process computes the output value. For instance, in the case of the execution unit from the running example we want to know how long it takes until a brake action is executed. Such delay is equal to the execution time of the execution unit process. Whether the execution time is equal to a computation time c strongly depends on the type of scheduling. If the system uses preemptive scheduling then the delay is equal to the relative deadline. Because the process can read the value at the beginning of the period, then it gets preempted and at the end of the relative deadline it produces an action. But DEECo does not specify scheduling details and it depends on specific implementation. Because of that we put the delay equal to the relative deadline.

During the calculation of a delay of a *producer* we have done implicitly quite a strong assumption. We have evaluated how long the value is not going to be updated by the process but that does not mean that another process that uses it will use it that late (the second part of our delay definition). Let us illustrate it on a theoretical example of three processes P_1, P_2 and P_3 that are scheduled in predetermined order. We define that P_1 returns the value that is used by P_2 and that returns the value used by P_3. We also add a guarantee that in each period P_3 uses a fresh value computed by P_2 in the same period and will also hold true for P_2 and P_1. The schedule of such setup is depicted in Figure 3.3.

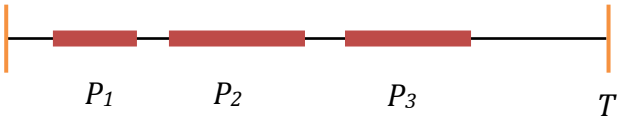


Figure 3.3: A schedule of strictly ordered processes P_1, P_2 and P_3

In Figure 3.3 there is no single value older than $\frac{1}{3}\,T$ and the total delay is lower than T. If we use the approach we have defined in the section above we would obtain a delay $2T + 3D$ which is quite a difference. In terms of finding the worst case we have not done anything wrong, the analysis just returns the worst estimation. However we should consider whether such approximation is still fine. The analysis is supposed to be used for systems that are based on the distributed communication and a scheduling of its processes is not that straightforward. For instance ensembles are executed parallelly over multiple components where it is quite hard to obtain such a nicely ordered schedule. For the sake of simplicity we stick with the estimation and allow the user to define custom delays for situations where such estimation is too imprecise.

The analysis could also take scheduling properties as an input but we want to stay with DEECo abstraction without the need to introduce new scheduling aspects. Such simplifications help to omit the requirement of doing simulation.

Triggered processes and ensembles

Another important part of the DEECo semantics is a class of triggered processes and ensembles. Triggered processes are triggered when a dynamic condition is met. The analysis cannot simulate such condition since it works with worst case estimations instead of exact values. However if the user wants to analyze a system which uses a triggered process and can guarantee that the process will be triggered then the analysis should handle it. A common usage of a triggered process can be for the emulation of interrupts.

We treat the condition of a triggered process as a black box which means we treat it in a way it can be triggered at any time. Because we are doing the worst case analysis we expect the process to be triggered in the worst possible timing. Such timing occurs when inputs are delayed as much as possible (such situation is depicted in Figure 3.4). The benefit the user has from a triggered process is its output delay is not affected by a period.

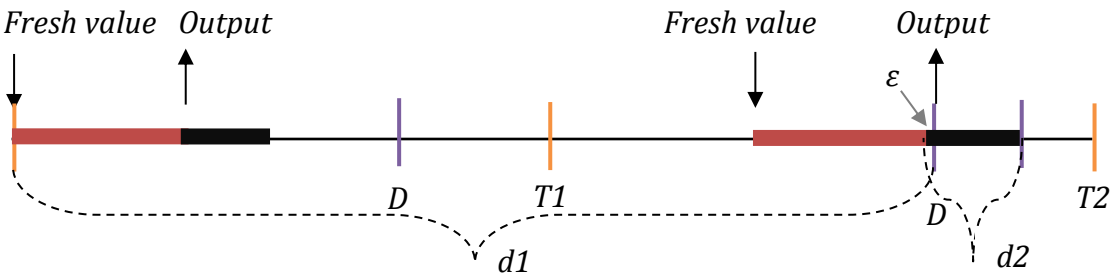


Figure 3.4: A schedule of the process from Figure 3.2 with an additional triggered process denoted by a black box and triggered ε before D.

There is a special case in which we overestimate the final delay. Such situation can occur when a process is triggered on the change of its input value. For instance, let us say that the control unit process is called only when speed is changed. Instead of delay $D + T$ it would be only D. So if a *producer* process returns a value that is used by the triggered process we should set its delay to D. This estimation works if the triggered process has only one input triggered. In the case of multiple triggered inputs it may happen that one input triggers change in situation when the other input is delayed in its worst case delay and vice versa. Because of this, for multiple triggered inputs we cannot set the delay to D only.

To summarize the discussion about triggered processes, in principle it is impossible for us to detect dynamic conditions of their triggers. In case the user can guarantee that the condition is met, it depends on the set of knowledge fields used in such condition. There is only one special case with a process having one triggered knowledge field which is also its input; in such a scenario the delay of the input knowledge field should be reduced to D. Because we do not want to complicate the tool with the evaluation of multiple vectors of values for multiple delays we do not optimize the special case. However the user can provide any process or ensemble with custom delays to optimize the results.

3.4. Scheduling and data flow order

In the previous chapter we have touched on the topic of scheduling. This poses a question on whether the analysis verifies the schedulability of a system.

The analysis is supposed to be used as an end tool for the evaluation of *obsolescence* of knowledge fields. The user must onward prove the schedulability of the system. What the user may not know and what is also the reason for usage of the analysis tool is information that some variable can get outdated. The user should provide the analysis with deadlines of each process and ensemble.

3.5. Dynamic behavior analysis

Will the dynamic behavior be covered by the analysis? By dynamic behavior we mean dynamic binding conditions of ensembles and dynamic triggers of ensembles and processes.

Dynamic ensembles interchange values of some particular subset of knowledge fields only for components which satisfy dynamic binding conditions defined over other subsets of knowledge fields. The problem is that even if we have the exact value of each knowledge field we cannot decide whether all these values will be present at the same time to satisfy its binding condition. This would require trying all scheduling possibilities to find whether an ensemble will be executed or not.

The problem is this would lead to exponential time complexity which is unacceptable due to our requirements. This means the analysis is able to analyze only a static snapshot of a dynamic system and the user has to guarantee a dynamic condition of a particular ensemble is met if the ensemble is part of the analysis input. This means dynamic ensembles must be mapped to static ensembles which are defined in the same way as processes, where we know in advance which knowledge fields are involved.

3.6. Intervals analysis

Does the analysis have to work with single values? Or can the user give it a range of values? Instead of giving the analysis an initial speed *130km/h* the user would give it a range *50-150km/h.* The analysis would then search for the worst case of the given range.

The analysis is supposed to be used only for a particular static snapshot of a dynamic RDS which is hard to achieve in the case of ranges of values. Because of that we do not include support for ranges of values but instead we allow the user to provide the analysis with vectors of values where such values will be processed independently but in a single run of the tool.

3.7. Required functions

In this part we focus on the discussion of what type of functions need to be implemented by the user. What we need is to calculate the upper and lower bounds based on a delay of each knowledge field that is an output of some process or ensemble.

For instance, we have a process which takes values of knowledge fields A_1 and A_2, and outputs a value of a knowledge field B. What we need is to take the lower and upper bounds of fields A_1, A_2 and transform them to the bounds of the field B and also count in the delay by which the field B was affected because of scheduling parameters of the process.

We have two options on how we can approach this problem. The first one is the user gives the analysis two functions. The first function would take

bounds of the fields A_1, A_2 and return bounds of the field B. The second function would then take bounds of the field B and delay and return bounds of the field B affected by the given delay. This means that to get the final bounds of the field B, the analysis would need to apply two functions.

The second approach requires only one function from the user. The function takes the bounds of fields A_1, A_2 and the delay and returns updated bounds of the field B where the affect by the given delay is already included. This looks like we are mixing together computations that should be completely separated. However there are a few benefits that outweigh this assumption.

The first benefit is that the user does not have to define two separated functions. Instead of simulating the real system the analysis works with its model. This means the required transformation of values from fields A_1, A_2 to the field B may not even exist in the real process because the analysis quite often works with a different set of variables. Hence the special function for values transformation (without taking a delay into account) may be misleading and confusing.

The last and most important problem of the first solution is that for a calculation of a field's value affected by the delay we may need to include other fields in the calculation. For instance, how the distance was changed strongly depends on the current speed. The user would need to supply the function with custom arguments representing other knowledge fields which start to become complicated and brings issues. For example the analysis could calculate the field value from different inputs rather than from which it calculates its delay.

The first option was previously implemented in the prototype and was finally changed to the second option because we find it more convenient. Meaning for every single process (and ensemble), the analysis requires a function that takes as arguments input knowledge fields of the process (or ensemble) and the delay and returns the value of the output knowledge field affected by the given delay.

3.8. Specific properties of evaluation functions

In the previous chapter, we have specified our requirements for the evaluation function. In this chapter we analyze what kind of properties an evaluation function should have.

Monotonicity

We require an evaluation function to be monotonous to be able to do the analysis correctly. The monotonicity of an evaluation function is in practice very common.

For a function that based on the given delay returns an upper estimation of a knowledge field, it makes sense that the estimation is greater when the delay is greater. For instance, when the vehicle keeps accelerating then the speed keeps increasing or reaches its maximum. A similar case would hold for a braking scenario when the speed would decrease to zero.

We can imagine a lot of physical and chemical processes can be characterized by a monotonous function. For instance the longer time we warm water, the warmer it gets. The same behavior is true for the relationship of heat and resistance. Can we even find an example of a function that characterizes a real process and is not monotonous so it can decrease its maximum after some time (or increase its minimum in the case of lower bound functions)?

There is a whole set of physical and chemical processes in the real world that even we are giving them constantly some input after some time they change their output from increasing to decreasing. This set of processes has one common denominator. So much input is given to these processes that they reach some maximum threshold and fail or revert. For instance if a boiler constantly boils water and produces steam then the pressure in the boiler will constantly grow until it reaches critical pressure. In the worst case the boiler blows up in a better case some security valve would decrease the pressure by releasing the steam. The function describing this process is definitely not monotonous.

Figure 3.5 below illustrates how the function representing the pressure in the boiler may look. The boiler warms the water with constant temperature, after some time steam is produced and the pressure keeps rising until the pressure reaches its maximum and the security valve is activated.

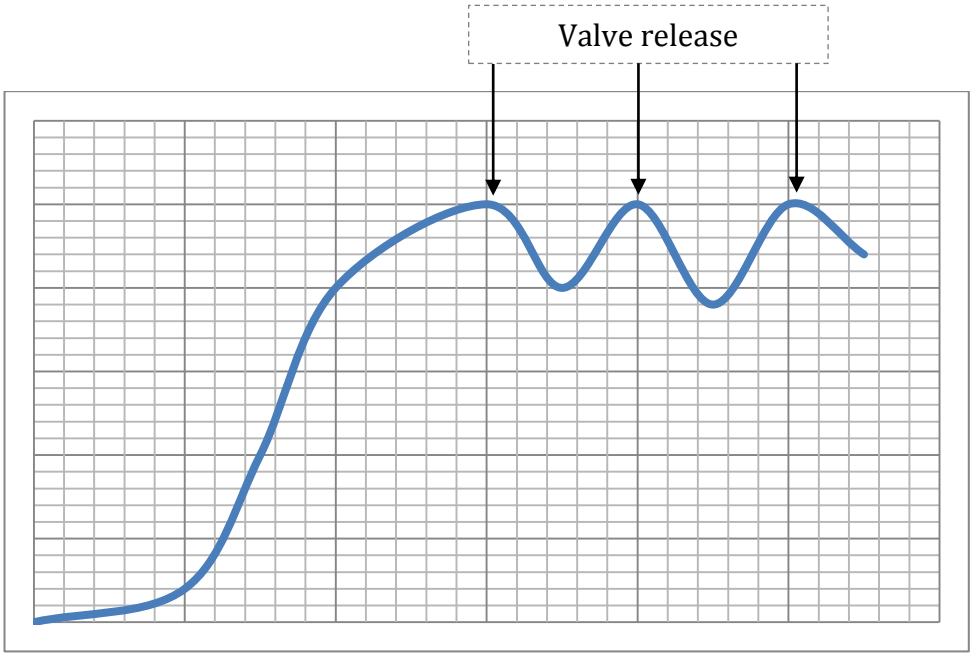


Figure 3.5: Abstract graph of a function representing steam pressure changes during valve release

The function of the pressure exactly describes a real process. The upper and lower bounds are represented by the maximum and minimum of the function. The maximal pressure is reached by boiling the water as much as possible. On the other hand the minimal pressure may be zero but it depends on the scenario. Therefore the function of the pressure does not suit the analysis well. The analysis needs a function that can answer the following question: "What maximum and minimum value was reached during a period between zero and the given delay?" This type of functions can be easily transformed to the required form. In Figure 3.6 below we see a graph of transformed upper bound function of a pressure.

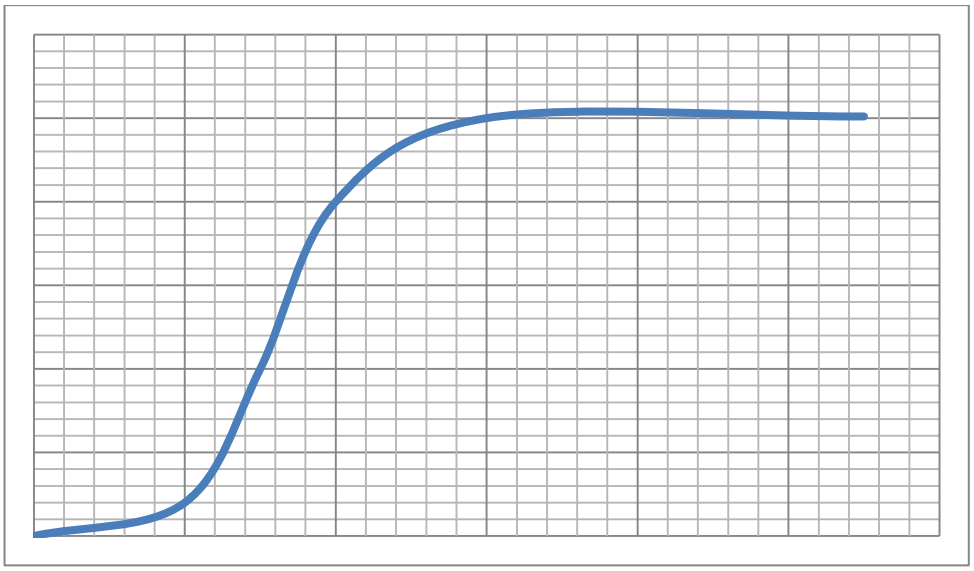


Figure 3.6: Abstract graph of the transformed upper bound function of steam pressure changes which masks the valve release

This means that the user must transform this type of function in a way that from the point when they reach their minimum or maximum they should be constant.

The second reason why we require monotonous functions is that during the analysis we need to guarantee that with increasing delay there is non-decreasing upper bound and a non-increasing lower bound. If not it would be hard to compose worst case estimations between multiple processes. Let us describe this in an example.

Let us say we have a process P_1 with a delay d_1 with an output field K_1. We set K_1 as an input of another process called P_2. Evaluation functions of both processes are monotonous so the result of the function P_2 strongly depends on the input of the process P_1 instead of just on a delay. The analysis tries to find the maximum of the function composed from two non-monotonous functions. What is worse is that the analysis does not know which knowledge field the user is interested in most. Is it a value returned by the last process in the chain? Or is it a value returned by the first process in the chain? We would need to transform these functions to monotonous by the same technique that we introduced in the first section when we were illustrating the example of the boiler.

Injective functions

We do not require evaluation functions to be injective since it does not have any impact on our analysis.

Continuous function

We do not require evaluation functions to be continuous. In fact it may be quite common that they are not continuous. For instance there is some threshold and after it is reached the system switches its state and the output rapidly changes.

Domain of a function

For the evaluation function f with input knowledge fields $K_1, \dots, K_n$ and delay d we define domain $D_f = \{(k1, \dots, kn, d) | \; ki \in D_{Ki};\; 0 \leq i \leq n;\;\; d \in \mathbb{N}_0^+\}$, where D_{Ki} is a domain of a knowledge field K_i. Domain of a knowledge field is defined in the DEECo semantics formalization.

3.9. Impact of scheduling effectivity on delay

Do we have to find the best possible scheduling to get the lowest possible delay to calculate the lower bound properly?

We do not because we search for the greatest possible delay of each process and ensemble. The reason is that we have two monotonous functions which take delay and affect upper and lower bounds. To obtain the lowest bound we need to use the greatest possible delay. As an example we can imagine actual speed. The lower bound is formed by a driver pressing the brake and the upper bound is formed by a driver pushing the accelerator to speed up. The longer the delay is the longer the brake is pressed and the lower the speed is.

3.10. Discrete knowledge fields

So far we have been taking into account only knowledge fields which contain numeric value. What about discrete values? A common type of a discrete knowledge field can be a field representing a state.

For instance, there is a semaphore that has *green*, *red,* and *orange* states. And we would like to know whether the system can "believe" that there is *green* on the semaphore but in reality there is *red* or *orange*.

The analysis does not care what the value represents. It only needs some initial value and evaluation function. What the user may find challenging is that discrete values like the ones representing states have no implicit information on how old they are. For instance the older the upper bound of speed is then the greater the value of speed. But what can the analysis do with knowledge that the state of the semaphore is *green*?

For instance the upper bound function can be designed in a way that it expects in the real world the state was read just few milliseconds before the change which means that it should be switched to the next state. The lower bound function would expect that the state was entered slightly before it was read. Therefore when the upper bound function gets *green* as the input it assumes that the state was switched to the *orange* state. If the delay is greater than the duration of the *orange* state it will jump into the *red* state. The lower bound function would treat the state new and if the delay is not greater than the duration of the *green* state it remains in that state.

Let us illustrate on a simple example what can go wrong with the approach mentioned above. There is an ensemble that transfers the state of the semaphore from one component to another affecting it with a delay of 100 *ms*. Then the value is used for a computation which affects it with a delay of 200 *ms*. Let us say that *orange*, *green* and *red* states have all the same duration equal to 250 *ms*. If the analysis would use the previously introduced evaluation functions for upper and lower bounds it would get as a final output: *green* -> *orange* -> *red* for the upper bound because every time we assume the state is old enough to be switched. From the lower bound function, we would get *green* -> *green* -> *green*. If we apply the function on the sum of those delays we would get as an upper bound result *green* -> *red* and for a lower bound result *green* -> *orange*. So for lower bound function the user receives a different output. This is logical since the composition forgets how long we are in that state which is incorrect.

Because of the issues mentioned above the better solution is to store in the knowledge field the delay of the semaphore instead of its state. The delay can be composed pretty easily without errors. The evaluation function would be only an aggregator summing those delays. At the end the user can use the final delay to decide what states can be reached via a timed automaton.

3.11. Evaluation function features analysis

We have decided to use one function for both the values transformation and delay effect calculation. In this chapter we discuss what the user must supply to the analysis and what the analysis can provide to help them. Let us say we have an ensemble called *SpeedEnsemble* which transfers value of a knowledge field *speed* to a knowledge field *speed2* and its delay is *d*. The user can then provide the analysis with the following function:

```
1. int speedEnsembleEval(int speed, int delay) {
2.  // calculate speed2
3. }
```

Source Code 3.1: Function for calculation of new value of speed based on the given delay.

The user needs to create a function that calculates a new speed from a given speed and delay. This may be a common task that will be necessary quite often. But the speed2 may be calculated from a differential equation because the derivation of speed is acceleration. The majority of such calculations is about how some value changes over time. If the analysis would get derivations of speed it could simply integrate it over a delay and get the final speed. What we need is a differential equation solver that takes input given as a table that contains values and their derivatives. This would downsize the problem for the user to just simply define the table instead of a need to design a whole solver. We also leave this possibility open if the user wants to customize the evaluation.

3.12. Data flow analysis

The technical aspect of the algorithm which we have to consider is a data flow. We presented the system as a set of knowledge fields bound together by processes and ensembles. But can these values be bound together arbitrarily or are there some restrictions?

If we represent bindings as an oriented graph we may be interested in cycles. That can happen when a process takes a knowledge field as an input and returns it as an output. Or in a more complicated way when a process A takes a knowledge K and returns K' and a process B takes knowledge K' and returns K.

The problem in cycles is not technical but semantic. What does this mean for *obsolescence* of a knowledge field which is used as an input and an output? The process A causes a delay to a knowledge field K' every time it is called. It also takes the delayed knowledge again as the input. This means that by definition the analysis should return an infinite delay. Even if there would be other process that would sometimes update the value it does not prevent the first process to always take its own output instead of the output of the other process because of some specific scheduling parameters. This observation leads us to completely forbid cycles in a knowledge path. The user may break a loop by creating a new knowledge field.

Another problematic binding is represented by two processes or ensembles which output the same knowledge field. We forbid this type of configuration since the DEECo semantics do not specify that it should be supported. The worst case estimation can be done by choosing the process with the maximal delay. The problem of this solution is that both processes can have completely different delay functions and the process with the lowest delay can return worse estimations than the process with the greater delay. This means that the analysis could not solve this in a general way and values comparators would be necessary.

3.13. Summarization

In this section we are going to summarize the results of the analysis chapter. We also introduce parts of the DEECo semantics which had to be simplified or even skipped.

Regarding the dataflow we have forbidden cycles between knowledge fields. We have also forbidden multiple processes to output the same knowledge field. In the context of runtime characteristics of DEECo we have omitted triggered processes and ensembles in a way that if the user wants to add a triggered process they must onwards know that it will be triggered and include it as any other basic process. We also cannot support dynamic ensemble conditions and their emulation must be done in the same way as triggered ensembles. In the context of knowledge fields we do not support structured knowledge fields and the user should be careful when passing in discrete knowledge fields.

We have defined what the term *delay* is and how it is calculated. To be able to analyze delay we distinguish on a theoretical level between the *consumer* and *producer* types of processes and ensembles.

As a user input we require besides the basic DEECo parameters an evaluation function for each output knowledge field of each individual process and ensemble. These functions have to be monotonous in respect to delay and the delay argument should be defined for all positive natural numbers.

Chapter 4

Algorithm overview

In this chapter we are going to introduce the main algorithm of the analysis.

Symbol $\mathbb{C}$ denotes a set of all components in the system. Symbol $\mathbb{E}$ denotes a set of all ensembles in the system.

For each knowledge field K_C of a component C we define D_{K_C} as a domain of the knowledge field K_C. A statement $K_C.values$ represents a set of bounds that the knowledge field K_C contains. $\mathbb{K}$ denotes a set of all knowledge fields of the system.

Each component C contains a collection of its knowledge fields in a property $C.knowledgeFields$.

For each process P_C of a component C we define $P_{C_{IN}}$ as a set of input knowledge fields and $P_{C_{OUT}}$ as a set of output knowledge fields.

1) Initialization

First, the algorithm assigns default vectors of bounds (given by the user) to their knowledge fields. If a values generator is assigned for some particular knowledge field then the generator is run and the returned vector of bounds is set as a default vector for that knowledge field. An overview can be found in Source Code 4.1.

```
1.  for (K_C in 𝕂) {
2.   if (exists generator G for K_C) {
3.    K_C.values = G.generateVector();
4.   } else if (exists vector V of default values for K_C) {
5.    K_C.values = V;
6.  }}}
```

Source Code 4.1: Initializes knowledge fields with default values.

2) Create oriented graph

The algorithm creates an oriented graph $G = (\mathbb{K}, \mathrm{E})$ where vertices will be represented by knowledge fields. For each process and its input and output knowledge fields we will add an oriented edge leading from the input field to the output field. The same will be done for each ensemble's input and output knowledge fields.

There are two assumptions that we need to cover. The first one is we can get vertices of the graph in a topological order. This is possible because we have forbidden oriented cycles in the analysis chapter. The second assumption is for every knowledge field there is only one process or ensemble that outputs it. This is also possible because we have forbidden multiple processes or ensembles to output the same knowledge field.

```
1.  for (P ∈ {P_C | P_C ∈ C.processes, C ∈ ℂ}) {
2.   for(K_IN in IN_P) {
3.    for(K_OUT in OUT_P) {
4.     E.add(K_IN, K_OUT);
5.  }}}
7.  for (E in 𝔼) {
8.   for(K_IN in IN_E) {
9.    for(K_OUT in OUT_E) {
10.     E.add(K_IN, K_OUT);
11.  }}}
```

Source Code 4.2: Creates an oriented graph from knowledge fields.

3) Traverse the graph

The algorithm traverses over the vertices (representing knowledge fields) in topological order and for each field that is represented by that particular vertex it takes processes or ensembles in which that field is used as an output. Then it applies the evaluation function defined in the next step.

```
1.  for (K in G.getVerticesInTopologicalOrder()) {
2.    if (K is output of process P_C) {
3.      evaluate (K, P_C);
4.    } else { // K is output of ensemble E
5.      evaluate (K, E);
6.    }
7.  }
```

Source Code 4.3: Traverses the graph in the toplogical order and calls evaluation functions for their processes and ensembles.

4) Evaluation function

The evaluation function takes a knowledge field K and a process P or an ensemble E of which the output is field K. For easier manipulation we define for the process and ensemble a common ancestor called *Task* denoted by a letter T.

First, the algorithm calculates the delay of the Task T. Because triggered processes and ensembles have a period equal to zero, a sum of the period and deadline can be applied. It will work for triggered and non-triggered tasks. If the algorithm works with the ensemble it counts in the belief propagation delay which is the overhead of belief propagation.

Because each knowledge field can have multiple bounds assigned the algorithm works with vectors of bounds. It takes all input vectors of bounds of all input fields and combines them to create a set of input vectors for an evaluation function. This can be done by a Cartesian product (if user requires) or by creating vectors from values with the same index. The set of these vectors is denoted by the symbol VC.

The next steps will be done for each vector of the VC set. Based on whether the user provides the analysis with an evaluation function or not we decide how to evaluate the output knowledge field. If the evaluation function is provided the algorithm calls it giving it the vector of bounds of the input knowledge fields and the delay. The function then returns bounds of the output knowledge field. Otherwise a differential solver is used to retrieve the bounds. Finally the result is appended to the final vector of values of the output knowledge. The whole procedure is then repeated for the next set of input bounds of the input vector VC.

1. public void evaluate(K, T) {
2. $delay = T.period + T.deadline$;
3. if $(T\ s\ an\ ensemble)$ {
4. $delay \mathrel{+}= beliefPropagationDelay$;
5. }
6.
7. $VC_1 = \{v_0 \times v_1 \times \ldots \times v_n \mid K_i \in IN_T, v_i \in K_i, |IN_T| = n\}$
8. $VC_2 = \{(v_0[k], v_1[k], \ldots, v_n[k]) \mid K_i \in IN_T, v_i \in K_i, |IN_T| = n, 0 \leq k < |K_i|\}$
9. $VC = VC_1\ or\ VC_2$ // depends on user configuration
10. if $(exists\ user\ defined\ function\ F)$ {
11. $K = \{F(v),\ v \in VC\}$
12. } else { // use differental solver
13. $DF = \{function\ F \mid K \in IN_T,\ \forall x \in D_K\colon F(x) = derivative\ of\ x\}$
14. $K = \{solveDiff(v, DF) \mid v \in VC\}$
15. }
16. }

Source Code 4.4: The evaluation function.

Note that the details of the differential solver are described in the prototype's documentation section in Chapter 5.

5) Results

At the end, the analysis has vectors of upper and lower bounds for each knowledge field so it can output the results.

4.1. Time complexity

We are going to evaluate a time complexity of the algorithm we have just introduced in the previous section. Let us say that n represents the number of knowledge fields in the system. We do not have to care about the number of processes or ensembles. There can never be more processes or ensembles than knowledge fields because the knowledge field can be output of only one process or ensemble. This is guaranteed by the analysis in Chapter 3. In the following section we estimate the time complexity of each part of the algorithm and finally present the time complexity of the whole algorithm.

We have to take into account that we work with vectors of values not scalars. The user may define large vectors of values and their Cartesian product may be exponential. Because of this we introduce symbol s_{def} which denotes the size of maximal vector of default values, formally $s_{def} = \max(\{|D_K|,\ D_K \in \mathbb{K}\})$.

The user may provide the analysis with custom functions and we have no control over their time complexity. Because of this we introduce symbol c_f which represents time complexity of the most time demanding custom function, formally $c_f = \max(\{O(f) |\ f\ is\ a\ custom\ func\ of\ the\ system\})$.

The last important part of the algorithm that affects the timing complexity is the differential equations solver which uses a step integrator. We define symbol c_i which denotes the amortized number of steps that the integrator does to solve the equation. This heavily depends on the type of a particular integrator, precision the user requires, and error allowance.

1) Initialization

Initialization of default values for all knowledge fields takes $O(n)$ steps where work done at each step is $O(s_{def})$. This gives us $O(n * s_{def})$.

2) Oriented graph creation + 3) Topological sort

The creation of the oriented graph with n vertices takes $O(n)$ steps. The algorithm for the topological sort of the graph edges takes $O(n)$ [7].

4) Evaluate

The *evaluate* function is executed at most n times. What we need to find is a timing complexity of the function.

An evaluation function runs over a Cartesian product of input knowledge fields' vectors. If each knowledge field has only one pair of bounds then the number of steps is $O(1)$ for each knowledge field. But if the analysis works with vectors of bounds then the complexity is $O(s_{def}^{n})$. In each step it calls an evaluation function given by the user or the differential solver. In the case of the user given function we have to count in its time complexity. The differential solver works in a way that in each step it requests a derivative of the value and calculates a new one. This means each time the solver requests the derivative the analysis does a binary search in a table of derivatives which takes $O(\log t)$, where t is the size of the table. The solver does c_i steps. This means that a single run of the evaluation function takes $O\big(s_{def}^{n} * c_i * (\log t_{max} + c_f)\big)$, where $t_{max} = \max(\{ t \mid size\ of\ a\ table\ t\})$.

Based on all the above criteria the final time complexity of the algorithm is $O(n * s_{def}^{n} * c_i * (\log t_{max} + c_f))$. This is exponential. However if the system's knowledge fields contain only a one pair of bounds and the complexity of each custom function is $O(1)$, which is somewhat expected, then the time complexity of the algorithm is $O(n * c_i * \log t)$ which is polynomial. This is what we wanted to achieve from the beginning. When we get a single evaluation of default values of knowledge fields of the system we are able to analyze it in polynomial time where the polynomial depends only on the number of knowledge fields.

Chapter 5

Prototype of the analysis tool

In this chapter we are going to describe a Java implementation of a prototype of the analysis tool that is part of the thesis. The source code can be also found on GitHub [8].

The analysis is organized into three projects. The first project is called *meta-model* and contains a meta-model that defines the semantics of a model that the user gives to the analysis as an input. Then there is a project called *generator* which takes a model defined by the user and generates interfaces that need to be implemented by the user and also bindings based on the configuration given in the model. The binding code contains code that wraps everything together and calls the analysis which is in a separate project called *analysis*. This means that the analysis project works directly with code entities. There is also an extra project *example* which contains the running example. The work flow of the tool is the following:

1) The user creates a configuration of a system. The configuration contains all knowledge fields, components, processes, ensembles and their necessary configuration. This is defined via model defined by the meta-model from the *meta-model* project.
2) The user then passes the configuration to the *generator* that creates the necessary Java classes including interfaces for ensembles and processes evaluation functions that need to be implemented. And also a binding code that will bind all things together and run the analysis. To do the analysis the user only has to run the generated method from their own project which references the analysis library.

3) The analysis then outputs its results to a standalone html file.

5.1. Meta-model

The *meta-model* project contains the meta-model which provides semantics for the definition of analysis input models. The meta-model is created in Eclipse Modeling Framework (EMF) [9] [10] and therefore it is recommended to use the EMF Eclipse edition which has all necessary built in tools for manipulation with the model.

The meta-model can be found in a file called AnalysisMetamodel.ecore in a *model* folder. For convenient manipulation we recommend to use a file called *AnalysisMetamodel.ecorediag* which offers graphical editor that shows the entities and their bindings in a clear graphical layout. All changes are reflected in the *AnalysisMetamodel.ecore* file. Image of UML of the meta-model is also attached on the CD.

From the meta-model we generate a Java source code which is used by the *generator* project. This code contains classes used to parse the final model that is given to the *generator* by the user. The file that is used to generate the meta-model code is called *AnalysisMetamodel.genmodel* and references the *AnalysisMetamodel.ecore* meta-model.

To define a model via the meta-model the user needs to open the *AnalysisMetamodel.ecore* file with an EMF editor, expand the nodes, click on the entity called *model* and select option "Create Dynamic Instance" from the context menu. This creates a *.xmi file that can be edited via Eclipse editor.

5.2. Code Generator

In this chapter we describe how the code generator, which is responsible for the generation of the code for the analysis, is organized.

The *Main* class takes arguments which define where the source code should be outputted, what package the generated classes should have, whether the implementation classes should be overridden, and a path to the input model from which the code will be generated.

At the beginning, the input model is checked with a validator represented by the *Validator* class. The validation is necessary because the EMF meta-model cannot reflect extended constraints such as duplicate names or some overloaded references between entities. For instance, we check that the model does not contain components or ensembles with the same name. We also check that the knowledge field references a generator that is defined for the same type. Last but not least we verify that for each type of knowledge its interval implementation is defined. If the validation is not done then the generated code would contain syntax errors and could not be compiled. The validator is not focused on constraints that are checked by the EMF validator.

After the input model is successfully verified the two generators *ClassGenerator* and *BindingGenerator* are called. The class generator is responsible for generation of standalone Java classes which includes definitions of abstract classes and their empty implementation stubs. The binding generator creates a glue code that binds all classes together, sets ups the analysis input entities, and runs the analysis located in the *analysis* project. Both generators are inherited from *AbstractGenerator*. The project also contains utility classes *GeneratorUtils*, *FileUtils* and *StringUtils*.

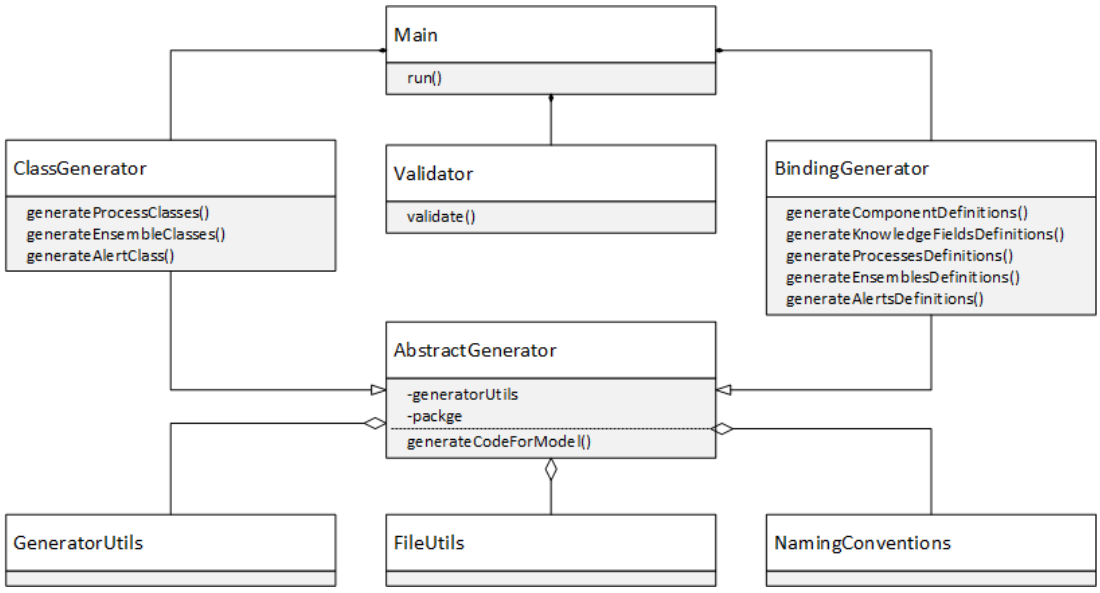


Figure 5.1: UML model of the main entities of the generator

There is also a class called *NamingConventions*. Its purpose is to unite names of generated classes and variable identifiers for all generators so the naming policy can be changed in one specific place.

For the code generation we use a template library called FreeMarker [11]. Templates can be found in *templates* folder.

5.3. Main analysis entities

In this chapter we introduce an overview of the *analysis* project together with an overview of its important classes.

The most important class is *Analysis*. This class starts the whole process. It creates the oriented graph and runs the analysis on every single knowledge field in topological order and prints results.

AnalysisConfiguration is a container class that contains all necessary definitions and configurations. There is a collection of all components (and their processes and knowledge fields) and ensembles. Everything is wrapped in a class and strongly typed. It represents the output the *Analysis* gets from the *generator*.

Each knowledge field is represented by a *KnowledgeField* class. This class contains some basic information about the knowledge fields such as a component's reference, name of the knowledge field, and other technical aspects that will be introduced later.

Numeric knowledge fields are a major part of the analysis input. They are used in differential equations, derivation tables, and also required for unit conversions. Because of this, a special successor of *KnowledgeField* class called *KnowledgeFieldNumeric* was created.

If derivations of the knowledge field are defined via the derivation table then a special successor of *KnowledgeFieldNumeric* class called *KnowledgeFieldViaTable* exists that has two extra references to a class *TableOfDerivatives* called upperBoundTable and lowerBoundTable.

Because the analysis works with upper and lower bounds and not with a single value, the abstract wrapper called *Interval* was introduced. The wrapper contains the bounds and provides additional operations for easier manipulation. The analysis requires the implementation of the *Interval* class for each type of knowledge field it works with.

Each process is represented by a *Process* class. Each ensemble is represented by a *StaticEnsemble* class. These classes both extend a class called *Task*. The reason is they have very common properties. Both static ensembles and processes are just some functions that transform input values to output values. The *Task* class just covers their common behavior. *StaticEnsemble* and *Process* both have to implement the method *getDelay* because they differ in the way they compute their delay. Ensembles must include the value of *beliefPropagationDelay* into the calculation. The last difference is that *StaticEnsemble* binds to a two components and *Process* binds to a single component.

Task references multiple instances of a class called *FuncWrapperBase*. The instance is defined for each output knowledge field and a particular subset of input fields to allow a calculation of upper and lower bounds based on the provided delay value. This means that when a process has output knowledge fields *A* and *B*, there will be two instances of *FuncWrapperBase* defined, one for the knowledge field *A* and the other for knowledge field *B*.

FuncWrapperBase class has a successor called *FuncWrapperDifferential*. The successor is used when an output is calculated from a differential equation set.

If there is a custom function defined by the user then the successor of *FuncWrapperBase* called *FuncWrapper* is used. Custom evaluation functions are wrapped with class *Func*.

There is also a class *Component* which represents a particular component of DEECo. Its purpose is just organizational.

Finally there is *AbstractOutputPrinter* class which prints the results of the analysis to the HTML file using a FreeMarker library for templates. The *AbstractOutputPrinter* takes *AnalysisConfiguration* as an input.

5.4. Analysis runtime flow

In this section we present the basic flow of the tool in the figure below.

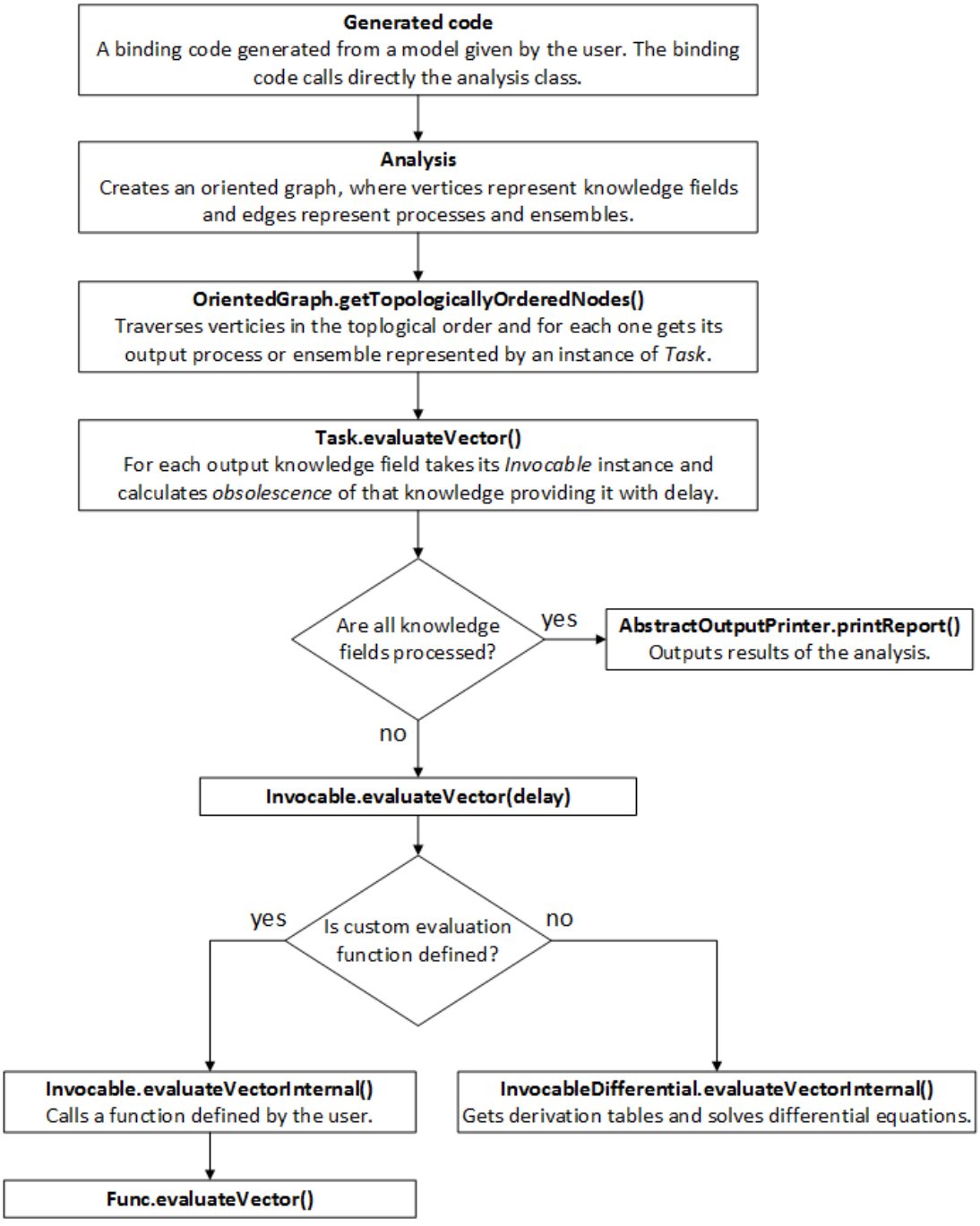


Figure 5.2: Flow of the protype tool

5.5. Differential equation solver

The calculation of a knowledge field's value based on its affection by a delay can be done via a built-in mechanism capable of solving differential equations combined with tables of derivatives. In this section we describe the basic principles of the process and how it is implemented.

Each table of derivatives is represented by *TableOfDerivatives* class. The class contains an array of sorted values and their corresponding derivations. In the meta-model semantics there is no way to force the user to give us the values sorted. Because of this there is a bubble sort algorithm that sorts the array of values and also updates the array of derivatives but only if the values were not properly sorted. When we need to obtain a derivation for a value that is not in the list of values we simply find the two closest values by binary search and do an interpolation defined by the equation as follows.

Interpolation is defined as: $\left[d(m) + \left(d(m+1) - d(m)\right) * \frac{value - v(m)}{v(m+1) - v(m)}\right]$, where $v(x)$ is a function which returns a value from the derivation table on position x and $d(x)$ is a derivation corresponding to the value $v(x)$. Index m is defined as follows $v(m) < value < v(m+1)$.

A table of derivations is assigned to a knowledge field for which it is defined. The class that represents such knowledge field is called *KnowledgeFieldViaTable*. The solver is aware of such type when using the knowledge field as an argument and automatically obtains derivations from the table.

Differential equations are solved using an iterative method based on temporal discretization for the approximation of solutions of ordinary differential equations. In the implementation we use a Dormand-Prince integrator [12] which is a member of the Runge-Kutta [13] family of ODE solvers. These are also called step integrators because they iterate via time steps over which they incrementally calculate the result. The implementation of the Dormand-Prince integrator is referenced from Apache Commons [14]. The class is called *DormandPrince853Integrator*.

However the analysis tool can work with any integrator that implements the *FirstOrderIntegrator* interface from Apache Commons. An integrator is passed to the analysis in the binding code defined by the generator.

Differential equations are solved using integrator solvers. These solvers work on principle where an integral is transformed into the difference of derivations divided by Δt; this operation is called temporal discretization. For the Runge-Kutta method the initial value problem is specified as follows:

$$y = f(t, y), \;\; y(t_0) = \; y_0$$

Where y is an unknown function (scalar or vector) of time t which we need to approximate. We also know that $\boldsymbol{y}$ is a function of t and of y itself. At the initial time t_0 the corresponding y-value is y_0.

When we have a function f which for arguments y_n and time t_n returns value y_{n+1} then we can calculate y step by step. The step depends on the type of the solver; it also depends on whether the solver is using a constant step or an adaptive step. However this is not a concern; what we have to do is to supply the function f to the solver.

Function f is represented by a method called *computeDerivatives* defined in the interface called *FirstOrderDifferentialEquations*. This interface is part of Apache Commons. We need to implement the interface and pass the implementation to the solver.

```
1.  void computeDerivatives(double t, double[] y, double[] yDot);
```

Source Code 5.1: Signature of a method from *FirstOrderDifferentialEquations*

Above is the signature of the method. Argument t represents a time t_n from the definition above. Array y represents the vector y_n. Array yDot will contain the result of the function f, the result of the method. This method is implemented in a class called *DifferentialEquation*.

Because the evaluation functions and their derivations are monotonous we do not need time information in our implementation. The vector y is just a set of values of knowledge fields which represent inputs of the evaluated

process or ensemble. This also means that the number of equations is same as the number of inputs which is also needed because we need the initial values.

When we have the vector of input values we simply iterate over these knowledge fields and check whether they are represented by a table of derivatives or by a custom differential equation.

In the case of the table of derivatives, we have already introduced its mechanism of finding such derivation based on given value. Therefore we only call a method *getDerivation* and pass it a value *y[i]* and store the result to *yDot[i]*. In the case of a custom differential equation we pass to the user's implementation whole array and a position *i* for which the computation has to be done.

This means that for a differential equation defined by the user we need to obtain some wrapper to which we can pass the arguments mentioned above. This wrapper is called *FuncWithEquationSet*. The method signature matches the arguments of the function *f*. If we would give such method to the user they would probably be lost in respect to which index represents which variable. For this reason the code generator provides the user with convenient wrappers that distinguish the knowledge fields by their name.

Below is a code snippet that demonstrates how the custom differential equation implementation can be wrapped by the code generator to make it clear for the user. This example calculates distance from speed. The user only needs to implement the function *getDistanceDerivation*.

```
1.  public abstract class Ensemble_Distance extends FuncWithEquationSet {
2.    @Override
3.    double F(double t, int i, double[] values, double[] valuesDerivatives) {
4.      if (i == 1) {
5.        return getDistanceDerivation(values[0], valuesDerivatives[0]);
6.      }
7.    }
8.
9.    double getDistanceDerivation(double t, double speed, double speedDe…);
10. }
```

Source Code 5.2: Example of a differential equation wrapper

At the end of the process we return the last item of the vector y as a result. We do this because otherwise we would need the user to implement the selector or to give us an index of the result.

5.6. Multipliers

We introduce multipliers because sometimes knowledge fields defined in different units are part of the same computation. This requires us to do conversions. Also it is more convenient to use human friendly units (e.g. m/s) when defining tables of derivatives and not be forced to use units that are used in computations (e.g. m/ms).

Multipliers can be found in the *KnowledgeFieldNumeric* class where such multiplier will cause all default values to be multiplied by its value and these values will be used in future computations. Values that are computed are not multiplied; in fact they are divided when they are about to be displayed in the output. Because the multiplier is used only for computations the data presented to the user should not be affected by it.

In *KnowledgeFieldNumeric* class we may see how the multiplier is applied. First, the default values have to be set before the multiplier. Otherwise these values will not be converted. Proper order is managed by the code generator. When the setter called *setMultiplier* is called all default values are copied into a special property called *valuesForTheOutput*. The previous values are transformed by the multiplier. When the output printer requests values to be printed the *getValuesForOutput* method is called. This method checks whether there are some values in a property *valuesForTheOutput*. If there are, they are directly returned. Otherwise the *values* property is taken and its values are divided by the multiplier.

Similar mechanisms exist for tables of derivatives which contain values and their derivatives. Both arrays sometimes need to be converted to different units. Because of this we introduce a values multiplier and a derivatives multiplier. When a values multiplier is set, all values are multiplied by it. When a derivative multiplier is set, all derivatives are multiplied by it. To the output we print original values and their derivatives (not affected by the multiplier).

5.7. Assertion mechanism

Because there are conditions that cannot be verified exactly at the time when something is being set up, there is an assertion mechanism that cares about the verification that all entities and bindings are properly set up. This serves mainly for us in situations when the code generator is not properly verifying all constraints or even generates a wrong output. A positive side effect is that we can be sure that invariants hold during the analysis and we do not need to check them later.

An assert mechanism is done in the way that each entity that requires its integrity to be checked implements an interface called *Assertable*. Each entity has to call method *assertConfiguration* on each contained entity which implements the interface *Assertable*. *AnalysisConfiguration* implements the interface *Assertable* and the method is called from the class *Analysis* on the beginning of the analysis process.

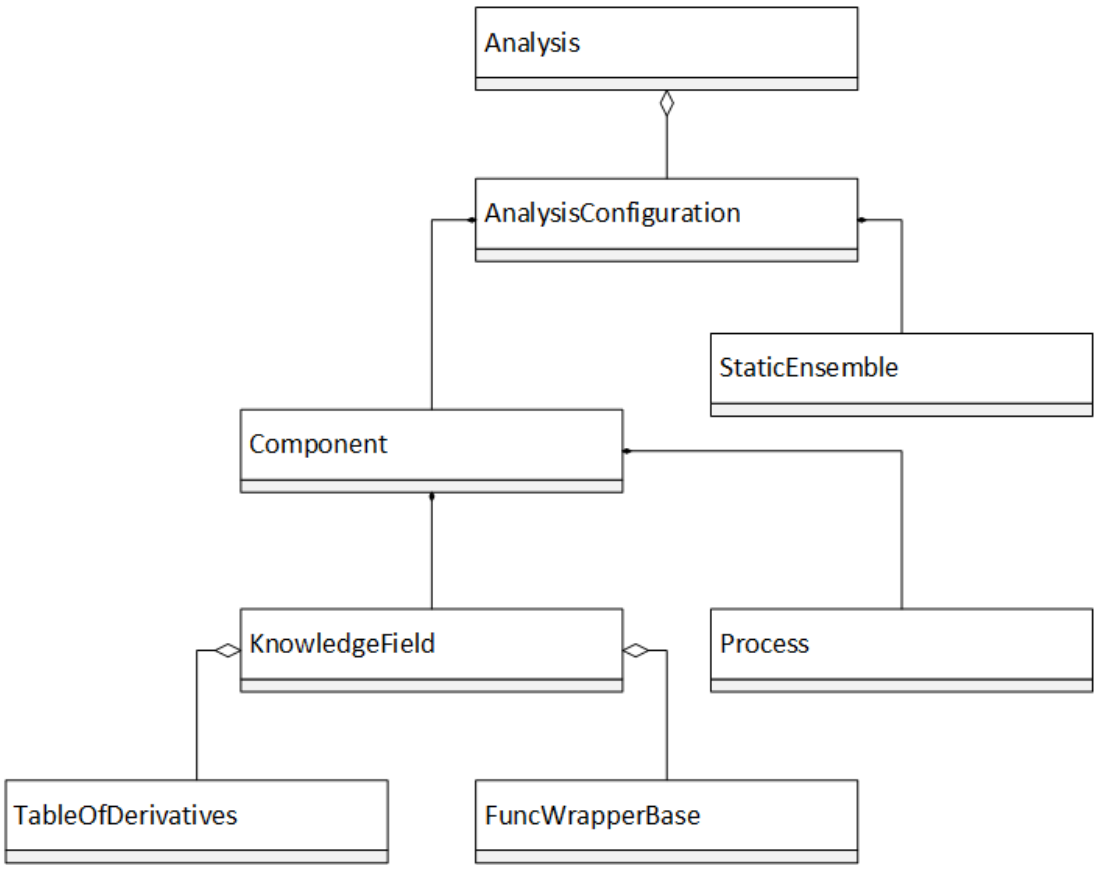


Figure 5.3: Call hierarchy of the *assertable mechanism*.

5.8. Customizations

In this chapter we describe what possibilities the user has to supply their own implementations of the specific parts of the analysis.

One of the advantages of the analysis is its complete independence on the types of knowledge fields it works with. When the user needs to evaluate some custom type they have to supply an implementation of its wrapper. This is done by inheriting the *Interval* class and adding its implementation reference to the analysis input model via the entity *CustomInterval*. The constructor must be the same as the constructor of *Interval* class.

Custom types may require custom value generators. The default built-in generator may also not be sufficient. In that case the user can supply their own generator implementation for a particular type of knowledge. The user can even supply multiple generators for the same knowledge type. This is done by setting an implementation reference to the analysis input model via the entity *CustomGenerator*. From there it may be assigned to a particular knowledge field that should have its default values generated.

The user may also provide the analysis with a custom step integrator. To do that they have to implement *IntegratorProvider* and set its instance directly via method *setIntegratorProvider* which is defined on *TimingAnalysis* class generated by the code generator. There is also a possibility to set up a custom analysis output printer if the user does not want to use the default one. This is done by implementing a class called *AbstractOutputPrinter*. The implementation is then passed to the analysis in the same way as the integrator via method *setOutputPrinter*.

5.9. Output export

Output is exported into an HTML file via *HTMLOutputPrinter* class that extends *AbstractOutputPrinter*. Output is formatted via a template engine called FreeMarker. Templates can be found in a folder called *templates*.

There is a template for results called result.html and a template errorResult.html that is used when something goes wrong.
The *HTMLOutputPrinter* uses *AnalysisConfiguration* class that holds all entities that were used during the analysis. From these entities the class mines out all necessary information for the output.

When the assertion mechanism finds some problem, it reports it to *ErrorReporter* class. This class is later passed to *AbstractOutputPrinter*.

Chapter 6

Evaluation

In this chapter we use the analysis tool to evaluate the running example introduced in Chapter 2. We are searching for situations in which the distance between two vehicles can reach a critical threshold. In the first part we define all necessary inputs of the analysis and then evaluate the results. The running example is also included with the source code in a project called *example*.

We define the initial speed values for both vehicles. Then we use the analysis to calculate the values of distance traveled by both vehicles and pass them through the whole system until they reach the execution unit. There we will have upper and lower bounds of distances travelled by each vehicle and we calculate the distance between those two vehicles. Because the analysis works only with static ensembles we have to define their bindings directly. Because the two vehicles will be close to each other we can safely assume that the ensemble membership predicate will be satisfied and there will be two instances of the ensemble running between them. One instance where the coordinator is the following vehicle and the second where the coordinator is the followed vehicle. Because the followed vehicle is not using information from the ensemble in our example, we define only one binding for the instance where the following vehicle is the coordinator. The ensemble will also interchange speed and travelled distance instead of position.

At the end each vehicle contains information about its own travelled distance and about the travelled distance of the second vehicle. We add a *Dummy* process with a zero overhead whose only purpose is to calculate the collision distance. The updated DSL of the example follows.

```
1.  component Vehicle:
2.      knowledge:
3.          initSpeed, initDistance
4.          mySpeed, mySpeedC
5.          myDistance, myDistanceC (travelled distance)
6.          fSpeed, fSpeedC
7.          fDistance, fDistanceC
8.          collisionDistance
9.      process ControlUnit:
10.         in mySpeed, in myDistance, in fSpeed, in fDistance,
            out mySpeedC, out myDistanceC, out fSpeedC, out fDistanceC
11.         scheduling: periodic(250ms), deadline(100ms);
12.     process ExecutionUnit:
13.         in mySpeedC, in myDistanceC, in fSpeedC, in fDistanceC,
            out myDistanceF, out fDistanceF
14.         scheduling: periodic(0ms), deadline(200ms);
15.     process Dummy:
16.         in myDistanceF, in fDistanceF, out collisionDistance
17.         scheduling: periodic(0ms), deadline(0ms);
18. ensemble UpdatePositionInformation:
19.   coordinator: Vehicle
20.   member: Vehicle
21.   membership: true
22.   knowledge exchange:
23.     coordinator.fSpeed = member.initSpeed;
24.     coordinator.fDistance = member.initDistance;
25.     coordinator.mySpeed = coordinator.initSpeed;
26.     coordinator.myDistance = coordinator.initDistance;
27.   scheduling: periodic(800ms), deadline(0ms);
28. system.beliefPropagationDelay = 200ms;
```

Source Code 6.1: Updated DSL of the running example to be used as an input of the analysis

First we should describe the changes we made in the DSL of the example. Let us take a look at the ensemble first. We have replaced the position knowledge with distance knowledge. What may look unexpected is that we made the ensemble update the coordinator's own speed and distance. We are doing this because the speed and the distance of the member will be affected by the delay of the ensemble. This means that the member vehicle already travelled some distance but the coordinator vehicle not. If we would be interested only in how the delay affects travelled distance of the followed vehicle for a static observer then this would not be necessary. Because we compare travelled distance of two moving vehicles, we also need to update the travelled distance of the following vehicle because during the ensemble run the following vehicle was also moving.

We have also extended the *execution* process so it computes speed and distance values because we want to find out what the real travelled distance will be from the time the followed vehicle sends us the information to the time the execution unit takes an action. The only way to find this out is to make the execution unit return values affected by its delay.

We have also assigned deadlines to each process and ensemble. We also had to set the period of the *execution* process to zero because normally the output of the process would not be used. However we added usage of the output by the *dummy* process and the analysis would evaluate the output with a different delay. This is covered in the analysis in Chapter 3.

As we have seen already the example could not be automatically transformed to the analysis input, because what we seek is not covered by the example definition.

As a next step, we need to define the evaluation functions for speed and distance. We define function *func_speed* that takes speed and delay and returns updated speed. This will be defined using a table of derivatives that will contain the acceleration for given speed values. The tables of derivatives are included in Appendix B.

We also need a function that will update the travelled distance. The travelled distance can be calculated from a delay, current speed and the initial travelled distance. We reuse our evaluation function for speed and add another equation for distance that will be identity (returning speed) because distance is the second derivative of acceleration. This function is called *func_distance*.

The functions *func_speed* and *func_distance* will be assigned to all processes excluding the *Dummy* process. The evaluation function for the *Dummy* process is defined as follows:

$$WorstCaseDistance.UpperBound = myDistanceC.UpperBound - fDistanceC.LowerBound$$
$$WorstCaseDistance.LowerBound = 0$$

Equation 6.1: An equation of the evaluation function of Dummy process.

We also need to define initial speed values for both vehicles. We will analyze the system with multiple initial speed configurations included in Table 6.1. The initial distance will be set to zero for both vehicles since it does not affect the meaning of the results.

$$initSpeed = (40, 50, 90, 100, 140, 150\ km/h)$$

Equation 6.2: Vector of initial speed (assigned to both vehicles)

When we set more than one default value to a knowledge field we have to be careful about evaluation. When the analysis works with such field it can create a Cartesian product of all input values or it can create vectors of values from each same index of all input vectors. The second option is possible only when each input field has the same number of values. For instance, when it calculates value of *fDistance* it has input from *initSpeed* with three values and from *initDistance* with one value. There the Cartesian product must be done automatically and *fDistance* will contain three values. However if we let it automatically calculate the *collisionDistance* the Cartesian product method would not be used. We do not want to compare only the same speed levels.

Because of this we set that we want to use the Cartesian method for the evaluation function wrapper.

The last thing we have to cover are units. Timing constraints of the system are defined in milliseconds, but values of the table of derivatives are defined in km/h and derivatives in m/s^2. The speed knowledge field is also defined in km/h but we want the distance knowledge field to be in meters.

The analysis tool has several settings to achieve the correct synchronization of units. The differential solver needs all tables of derivatives to have the same units as the scheduling. Because the example is in milliseconds we have to convert the values to km/ms and their derivatives to km/ms^2.

The differential equation solver would need km/ms as an input and would return km, but we would like to have the distance in meters. This means that we transform the values to m/ms and their derivatives to m/ms^2. This will also require converting speed to m/ms units.

To achieve this we need to set multipliers for tables of derivatives. For values we need to convert km/h to m/s. This means we divide the values by $\frac{60 \times 60 \times 1000}{1000} = 3600$. For derivatives we need to convert m/s^2 to m/ms^2. This is done by dividing the derivatives by 1000×1000.

For the distance knowledge field we don't need to do any conversions since the derivation table is defined in meters.

For the speed knowledge field we need to convert its units from km/h to m/s by dividing by 3600. This means that initial values will be accepted and printed in km/h but for the computation they will be converted to m/s. We also need to set this multiplier for any other speed knowledge field even though we do not provide them with default values. Otherwise values of those knowledge fields would be printed in m/s.

The defined input model can be found in the example project in a file called model.xmi. An overview can be found in Figure 6.1. For detailed

configuration we recommend to browse the actual model which is attached with the *example* project on the CD.

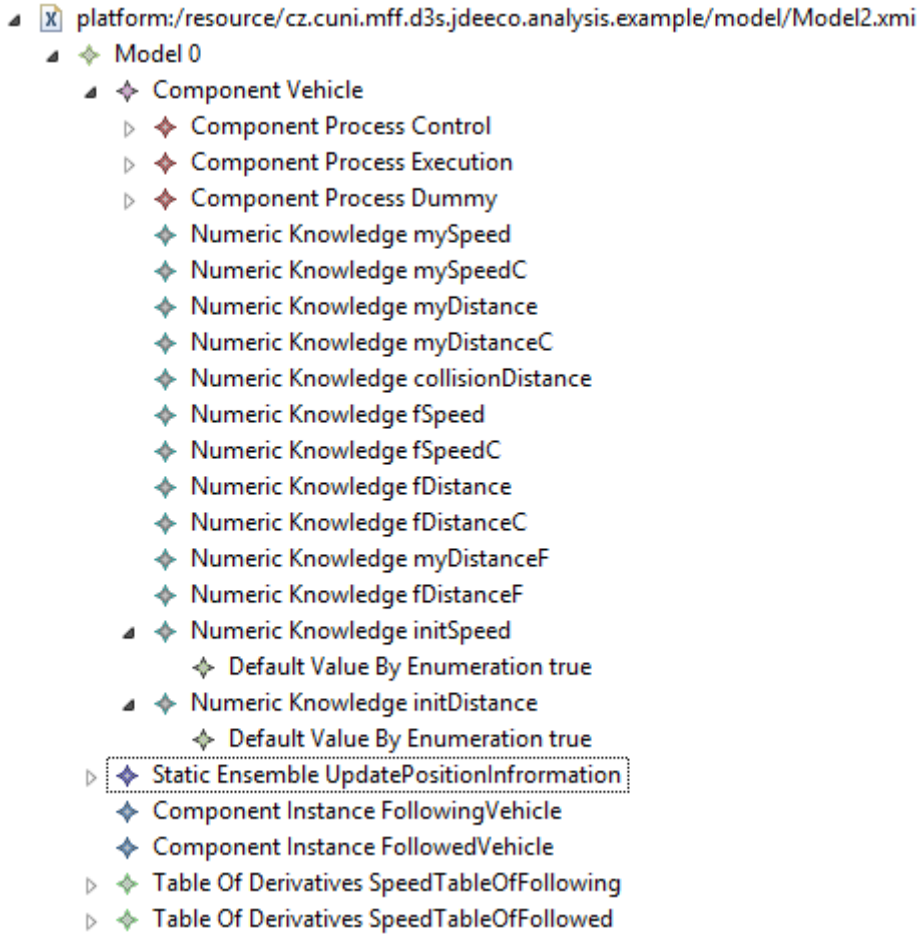


Figure 6.1: A model of the running example to be used as an input of the analysis

If we want to create a new project we only need to reference the analysis library jar attached on the CD as cz.cuni.mff.d3s.jdeeco.analysis-0.0.1.jar.

To run the code generator the user has to navigate to the *generator.jar* and provide it with required arguments. Template of the command can be seen in Command 6.1.

```
> java -jar generator.jar -m "path to the model" -o "output directory" -p
  "package name"
```

Command 6.1: A template of a command to be used to run the generator tool

Below we present a command that was used to generate the code for the *example* project which comes with the source code of the prototype.

```
> java -jar generator.jar
  -m "C:/…/cz.cuni.mff.d3s.jdeeco.analysis.example/model/Model.xmi"
  -o "C:/…/cz.cuni.mff.d3s.jdeeco.analysis.example/src-gen"
  -p cz.cuni.mff.d3s.jdeeco.analysis.example
```

Figure 6.2: A sample of the command used to generate code of *example* project.

When we have generated the code we need to supply the implementation of methods of classes located in the "*impl*" package. In most cases we have to supply the implementation of differential equation provider for the calculation of distance. This is done by returning a value of a knowledge representing speed. For the collision distance process we return a result based on the function defined previously. Details can be found in the example code that is attached on the CD.

Finally we define our code from where we instantiate the TimingAnalysis class and call the *run* method. When everything is set up we can run the analysis. The results are found in the HTML file called "test.html".

When we inspect the results we find that there are some values for *collisionDistance* that we do not know from which initial speeds they were calculated. It is hardly possible to trace back to the initial speed values. Because of this the tool has a pivot mechanism. We can mark important knowledge fields as pivots. Then during the computations, these pivots are transitively passed to all knowledge fields which uses their values or values calculated from their values. Because of this for each value we can simply track from which pivot values it was calculated. To do this we just set each such knowledge field to show its pivots in the output. To summarize the changes, for *initSpeed* we set a property *isPivot* to *true*, and for *collisionDistance* we set *showPivots* to *true*. Now we can run the analysis again and check the results.

6.1. Results

Results are standardly saved in the execution directory in a HTML file called results.html. If the user would like to change the directory they can do so by providing the analysis with *HTMLOutputPrinter* with a different path.

Because we have set up *initSpeed* knowledge field as a pivot and *collisionDistance* field to display its pivots we may directly inspect speed values of both vehicles and their effect on the collision distance. Important results can be found in Table 6.2 below.

Initial speed of following vehicle (km/h)	Initial speed of followed vehicle (km/h)	Collision distance (meters)
150	150	17,4
140	140	17,8
140	150	12,4
150	140	22,8
100	100	20,7
90	100	14,2
100	90	26,1
50	50	21,8

Table 6.2: Results of the running example

The greatest value comes from a situation where the following vehicle is approaching with speed 100 km/h and the followed vehicle has speed 90 km/h. When the followed vehicle starts to brake immediately and sends those values via ensemble to the following vehicle, the distance between these two vehicles can decrease by at least 26 meters until the following vehicle gets new data and executes braking instructions. That is for an example length of six average cars.

Because such distance already leads to collision it is not sufficient and the safety distance could be half larger. This is not very efficient in dense traffic. An interesting finding is that the situation is almost the same for speed around 50 km/h where 22 meters distance between vehicles seems too much. What a designer of such system has to do is to introduce other mechanisms or try to decrease the delays of the involved processes and ensembles.

Chapter 7

Related work

In this thesis we focus on the timing analysis of *obsolescence* of variables in resilient distributed systems. An interesting finding is in context of RDS or embedded systems generally, nobody has published any research in the area of *obsolescence* of variables.

If we focus on timing analysis of embedded systems in general, we may find intensive research in the area of hard real-time systems where providing runtime guarantees is necessary [15]. In such systems the engineer searches for the worst case execution time of the system's tasks and based on this they do a schedulability analysis. Finding of worst case execution times involves a code analysis with necessary guarantees from the user. Because such analysis cannot be precise without the knowledge of the underlying architecture, they introduced algorithms for a processor analysis [16] [17] including a cache analysis [18] and a memory simulation; they even take into account delays based on increased temperature of the environment during a processor workload [19]. A pain for developers are interrupts or parallel processing. Previous research did not bring complete solutions that would run in polynomial time for parallel systems. Hence, developers try to design such systems as simple as possible with a minimal level of parallelism, interrupts, and abstractions of hardware. This goes against parallel behavior of systems defined via DEECo.

However, the evaluation of *obsolescence* of a variable requires primarily calculating the longest possible delay for which the variable is not updated. We can focus on research in an area of identifying such delays.

We can imagine this task as a graph where vertices represents knowledge fields and edges represents time properties and we try to figure out how long it takes to get from one vertex to another. This approach looks similar to timed automata [20]. Timed automata are quite often used in model checkers such as PRISM [21] [22] which is designed for probabilistic models. There is also a model checker called UPPAAL [23] [24] which supports parallel automatons. Another model checker that uses *Timed* CTL (TCTL) [25] which allows us to define formulas with time conditions. These model checkers are used for verification of real-time systems. What we would need to do is define timed automaton for each component and ensemble since they all run in parallel and then do their composition and define some formulas (TCTL) to be verified. For this task we could use UPPAAL. This model checking approach is great when we want to guarantee some assumptions about the system. We can check whether the system is schedulable or if one action always follows another. We can even set time constraints between a variable's write and read operations to verify the maximal delay a value can have [26]. This is not quite suitable for our needs because we work with abstraction where all processes run in parallel and are black boxes for us. We also need to compute the delay values rather than verify their correctness. Time complexity of such solution would also be exponential which is not feasible for our analysis.

The term *belief* is also used in BDI agents [27]. BDI stands for Belief Desire Intensions. These agents are used in artificial intelligence. Their function strongly depends on a set of logical formulas from which new information are derived using logical formulas operations. There was a paper [28] published where the degrees of belief are used in a way that each formula has some degree of certainty and this degree is preserved and transformed during derivation. The final derived formula has assigned a degree that determines how reliable the information is. This solution is for problems represented by Boolean variables with values defined by a probability. What we would need is to support real numbers with time slots and a probability. Unfortunately belief outdating (causing change of degree of certainty) due to time delays was not researched in this area, based on our best knowledge.

If we focus only on searching for delays we may try to explore other areas where delays are analyzed. One such area is a timing analysis of circuit's gate delays. In this area a worst-case static timing analysis was overrun by a probabilistic (statistical) timing analysis (STA) because the static analysis overestimated the delays [29] [30]. In STA uses a technique that uses incremental searching of paths in a statistical timing graph [31]. Some solutions also include the calculation of lower and upper bounds and subsequent evaluation of their quality [32]. These solutions run in polynomial time in the number of gates. However these algorithms are specialized for tasks where probability is heavily involved.

Chapter 8

Conclusion

The goal of the thesis was to design timing analysis over the DEECo semantics. We have analyzed what should be the expected output and what type of information the analysis needs (including implicit guarantees made by the user).

At the end of the design process we have defined the core algorithm of the analysis that can be found in Chapter 4. We have achieved polynomial time complexity by doing a complete analysis of the required elements.

In the beginning, we expected that we can take the DEECo model and give it as an input of the analysis. We realized that the analysis needs a lot of specific information the general model cannot provide. The input task defined via the model sometimes must be redesigned so it can be analyzed in the way we want. This also led us to define our custom meta-model which is used for defining input of the analysis.

We have also implemented a prototype tool that is documented in Chapter 5. The tool was successfully tested on the running example introduced in Chapter 2 and confronted with results in Chapter 6. To make the tool complete we have also introduced the code generator that is able to (from the input model mentioned above) generate necessary binding code and allow the user to add their own implementation of the evaluation functions in a convenient way.

8.1. Future work

In the design process we have done a few restrictions such as omitting dynamic ensembles conditions, triggered processes and ensembles or discrete values. We let the user decide whether a dynamic condition in a particular configuration holds or not. Details of all restrictions and design decisions can be found in the Conclusion section of Chapter 3. An interesting future work should be on how the analysis of the dynamic behavior can be improved.

Future research can also be done with structured knowledge fields. There is a question on how to evaluate their values based on the affected delay. Whether we should evaluate each primitive field of the structured field alone and that structured field will be just namespace or whether we should analyze it as a whole.

Literature

[1] T. Bures, I. Gerostathopoulo, P. Hnetynka, J. Keznikl, M. Kit, F. Plasil. DEECO: an ensemble-based component system. *Proc. CBSE '13.* New York : ACM, 2013. ISBN 978-1-4503-2122-8.

[2] J. Keznikl, T. Bures, F. Plastil, I. Gerostathopoulos, P. Hnetynka, N. Hoch. Design of Ensemble-Based Component Systems by Invariant Refinement. *Proc. CBSE '13.* New York : ACM, 2013. ISBN 978-1-4503-2122-8.

[3] R. Al Ali, T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, F. Plastil. DEECo Computational Model - I. Charles University in Prague, D3S, 2013.

[4] J. Barnat, N. Benes, T. Bures, I. Cerna, J. Keznikl, F. Plasil. Towards Verification of Ensemble-Based Component Systems. *Proc. of FACS '13.* Springer, 2013. ISSN 0302-9743.

[5] M. Kit, I. Gerostathopoulos, T. Bures, P. Hnetynka, F. Plasil. An Architecture Framework for Experimentations with Self-Adaptive Cyber-Physical Systems. *Proc. of 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems.* Florence, 2012.

[6] G. C. Buttazzo. Hard Real-Time Computing Systems. Springer, 2011. ISBN 978-1-4614-0675-4.

[7] Wikipedia. Topological sorting. *Wikipedia.* [Online] 2015. http://en.wikipedia.org/wiki/Topological_sorting.

[8] F. Pavlis. Public repository of the DEECo Timing Analysis. *GitHub.* [Online] 2015. https://github.com/CodePhill/DEECo-Timing-Analysis.

[9] The Eclipse Foundation. Eclipse Modeling Framework. *Eclipse.* [Online] 2015. http://www.eclipse.org/modeling/emf/.

[10] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks. EMF: Eclipse Modeling Framework, 2nd Edition. Addison-Wesley Professional, 2008. ISBN 978-0-321-33188-5.

[11] A. Szegedi, D. Dekany, J. Revusky. *FreeMarker.* [Online] 2015. http://freemarker.org/.

[12] Wikipedia. Dormand–Prince method. [Online] 2015. http://en.wikipedia.org/wiki/Dormand-Prince_method.

[13] Wikipedia. Runge–Kutta methods. [Online] 2015. http://en.wikipedia.org/wiki/Runge-Kutta_methods.

[14] The Apache Software Foundation. Apache Commons. [Online] 2015. https://commons.apache.org/.

[15] S. Malik, M. Martonosi, Y. S. Li. Static Timing Analysis of Embedded Software. *Proc. DAC '97.* New York : ACM, 1997. ISBN 0-89791-920-3.

[16] S. Bharrat, K. Jeffay. Predicting Worst Case Execution Times on a Pipelined RISC Processor. Chapel Hill : University of North Carolina, 1995.

[17] T. Mitra, A. Roychoudhury, X. Li. Timing Analysis of Embedded Software for Speculative Processors. *15th International Symposium on System Synthesis.* Kyoto : IEEE, 2002. ISBN 1-58113-576-9.

[18] Ch. G. Lee, J. Hahn, Y. M. Seo, S. L. Min. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers.* New York : IEEE, 1998. ISSN 0018-9340.

[19] S. Wang, Y. Ahn, R. Bettati. Delay Analysis in Thermal-Aware Hard Real-Time Computing.

[20] R. Alur, D. L. Dill. A theory of timed automata. *Theoretical Computer Science.* Essex : Elsevier Science Publishers, 1994. Vol. 126. ISSN 0304-3975.

[21] A. Hinton, M. Kwiatkowska, G. Norman, D. Parker. PRISM: A Tool for Automatic Verification of Probabilistic Systems. University of Birmingham.

[22] University of Oxford. *PRISM Model Checker.* [Online] Department of Computer Science, University of Oxford, 2015. http://www.prismmodelchecker.org/.

[23] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, W. Yi. UPPAAL — a tool suite for automatic verification of real-time systems. *Hybrid Systems.* Springer, 1996. Vol. III. ISSN 0302-9743.

[24] G. Behrmann, A. David, K. G. Larsen. A Tutorial on Uppaal. Springer, 2004. ISSN 0302-9743.

[25] R. Alur, C. Courcoubetis, D. Dill. Model-Checking in Dense Real-Time. Stanford University. 1993.

[26] A. Alfonso, V. Braberman, D. Garbervetsky, N. Kicillof, A. Olivero, F. Schapachnik. VInTiMe: Combining High-Level Finesse with Low-Level Muscle to Verify Real-Time Systems. *Proc. of PRISE '04.* Buenos Aires, 2004.

[27] A. S. Rao, M. P. Georgeff. BDI agents: From theory to practice. *Proc. of the First International Conference on Multiagent Systems.* San Francisco : The AAAI Press, 1995. ISBN 9780262621021.

[28] S. Parsons, P. Giorgini. An approach to using degrees of belief in BDI agents. *The Springer International Series in Engineering and Computer Science.* Springer, 2000. Vol. 516. ISBN 978-1-4613-7373-5.

[29] M. Orshansky, K. Keutzer. A general probabilistic framework for worst case timing analysis. *Proc. of Design Automation Conference.* Washington : IEEE, 2002. Vol. 39. ISBN 1-58113-461-4.

[30] D. Blaauw, K. Chopra, A. Srivastava, L. Scheffer. Statistical Timing Analysis: From Basic Principles to State of the Art. *IEEE transactions on computer-aided design of integrated circuits and systems.* Washington : IEEE, 2008. Vol. 27. ISSN 0278-0070.

[31] H. Chang, S. S. Sapatnekar. Statistical timing analysis considering spatial correlations using a single PERT-like traversal. *Proc. of the 2003 IEEE/ACM international conference on Computer-aided design.* Washington : IEEE, 2003. ISBN 1-58113-762-1.

[32] A. Agarwal, D. Blaauw, V. Zolotov, S. Vrudhula. Statistical Timing Analysis using Bounds and Selective. *IEEE transactions on computer-aided design of integrated circuits and systems.* New York : IEEE Circuits and Systems Society, 2003. ISSN 0278-0070.

Abbreviations

BDI Belief Desire Intensions

CTL Computation Tree Logic

DSL Domain-Specific Language

EBCS Ensemble Based Component Systems

EMF Eclipse Modeling Framework

RDS Resilient Distributed System

TCTL Timed Computation Tree Logic

A. Contents of attached CD

The thesis has attached a CD and a USB flash drive (for the ones who already thrown away their CD mechanic), description of the directory tree follows. The source code of the prototype is also publicly accessible on GitHub [8].

/prototype – *contains the source code*

- cz.cuni.mff.d3s.jdeeco.analysis-0.0.1.jar – *the analysis library to be referenced by other projects that need to run the analysis*
- generator.jar – *executable jar file containing the generator of a source code which is necessary to run the analysis*
- cz.cuni.mff.d3s.jdeeco.analysis – *analysis project*
- cz.cuni.mff.d3s.jdeeco.analysis.generator – *generator project*
- cz.cuni.mff.d3s.jdeeco.analysis.metamodel – *project with the meta-model*
- cz.cuni.mff.d3s.jdeeco.analysis.example – *running example project*

/prototype-docs – *contains generated java doc*

- analysis – *java doc of the analysis project*
- generator – *java doc of the generator project*
- AnalysisMetamodel.png – *an Image of UML of the meta-model*

/results/exampleResults.html – *the results of the running example*

/Thesis.pdf – *the digital version of the Thesis*

B. Running example attachments

The acceleration table for both cars

The following car		The followed car	
Speed (km/h)	**Derivation (m/s^{-2})**	**Speed (km/h)**	**Derivation (m/s^{-2})**
0	9,00	0	9,00
10	6,50	10	6,50
20	4,90	20	4,90
30	5,60	30	5,60
40	4,40	40	4,40
50	5,25	50	5,25
60	2,90	60	2,90
70	2,80	70	2,80
80	2,70	80	2,70
90	1,72	90	1,72
100	3,00	100	3,00
110	3,00	110	3,00
120	1,44	120	1,44
130	1,90	130	1,90
140	1,45	140	1,45
150	1,25	150	1,25
160	0,96	160	0,96
170	0,82	170	0,82
180	0,58	180	0,58
190	1,00	190	1,00
200	0,60	200	0,60
210	0,30	210	0,30
220	0,15	220	0,15
230	0,00	230	0,00

The braking (negative acceleration) table for both cars

The following car		The followed car	
Speed (km/h)	**Derivation (m/s^{-2})**	**Speed (km/h)**	**Derivation (m/s^{-2})**
0	0	0	0
1	-8	1	-8
230	-8	203	-8